Telecommunications Service Access and Subscription (TSAS) Specification

DTC/2002-04-02

Final Adopted Specification
April 2002

X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Associated OMG Documents

The CORBA documentation set includes the following:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBA Languages*, a collection of language mapping specifications. See the individual language emapping specifications.
- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.
- *CORBAfacilities: Common Facilities Specification* includes OMG's Common Facility specifications.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Med*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404

Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Alcatel
- AT&T
- Britich Telecommunications plc.
- Cicso Systems
- Deutsche Telekom AG
- GMD Fokus
- Hitachi
- Humboldt University
- IBM Telecommunications Industry
- KPN Royal Dutch Telecom
- Lucent Technologies
- Nippon Telegraph and Telephone (NTT) Corporation
- Nortel Networks
- Siemens AG
- Sprint
- Sun Microsystems

Description 1

This chapter introduces the key concepts used in this specification.

Contents

This chapter contains the following sections.

1.1 Motivation

Network operators have traditionally followed a network-centric approach to delivering scalable, reliable and economic services to consumers and enterprises. The basic functions that are required to support services such as 800 numbers, call waiting and personal numbering have been under the exclusive control of the network operators. Enterprises and service providers wishing to offer value-added solutions, such as call centers, have had to rely on an edge-of-network approach and have been denied access to useful information and capabilities within the network.

The disadvantages of this separation are significant in today’s marketplace. Network operators employing a network-centric approach are unlikely to have the resources and flexibility necessary to respond to the specialized requirements of different customer

markets. Similarly, solution providers adopting an edge of network approach, while they may have the flexibility required for customizing services, are unable to gain the efficiency of using in-network functions and information. The architecture of the Telecommunication Service Access and Subscription (TSAS) specification combines the benefits of the network centric approach of economies of scale with the flexibility of the edge of network approach.

The set of interfaces contained within this specification provide the domain facilities through which network operators can offer 3rd party enterprises secure access to the capabilities of the network. Capabilities such as call control and user location can be offered (through their own interfaces) or by 3rd party value-added services and solutions.

Of course this approach is not only applicable to providing access to embedded network capabilities. It can also be used for a wide range of commercial models supporting customer-to-business or business-to-business relationships for eCommerce and the Application Service Provider market in general. Provision of functions for billing and payment can be easily integrated.

It is not within the scope of this specification to restrict the breadth of [component] services that could be offered by TSAS. This specification is technically aligned with that of the Parlay Group [Ref http://www.parlay.org]. Consequently the service interfaces specified in the Parlay API [Parlay API specification 2.0] can be offered using this specification.

1.2 Roles and Domains

Three different domains are defined for TSAS as shown in Figure 1-1: Consumer Domain, Retailer Domain, and Service Provider Domain.

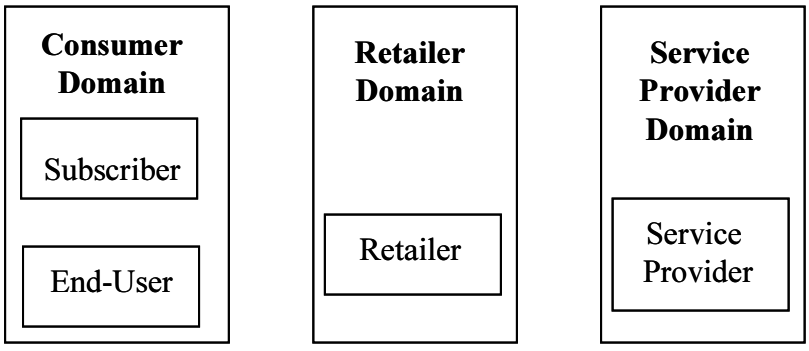

Figure 1-1 TSAS Domains

The domains are strongly correlated to *roles*, which will be explained in the following text.

In the *Consumer Domain* two kinds of roles are defined, the *end-user* role and the *subscriber* role. Typically end-users can be private households or any kind of company. The *end-user* is the one that makes use of the service while the *subscriber* holds the contract with the retailer and subscribes to services for its users. This can be depicted with a very common example: A company - the subscriber - has a subscription contract with a telephony provider. In the contract the rights of the different employees are defined - the employees are the end-users. In the case of a private household the subscriber and end-user role are identical.

Within the Retailer Domain the retailer role is defined. The *retailer* provides an integrated view of services to the end-user or subscriber. A major point of value added services offered by retailers is the unified management of services, in particular in terms of subscription facilities. Retailers thus act as middlemen for service providers and present a single point of contact to end-users and subscribers. This is an analogy to the notion of one-stop-shopping in a supermarket. Retailers have to ensure the ease and quality of service access.

Within TSAS the retailer is giving end-users a single point of contact for all their service needs. Additionally, the retailer enables end-users to customize and personalize services that they use by providing facilities to configure and select services incorporating personal preferences.

A prerequisite for service provisioning is a contractual relationship between service providers and the retailer. For service access a contractual relationship must exist between the subscriber and the retailer. No direct contractual relationship is required between end-user and service provider since the retailer mediates between both.

In general, the retailer:

- manages contracts for end-users and service providers,
- locates matches between user requirements and service provider subscription offers, and finally,
- enables the interaction between end-user and service provider.

In the Service Provider Domain the *service provider* role is defined. It offers its services to the end-user (or subscriber) through a retailer, or in other words, it supports the retailer with services. In addition, the retailer allows service providers to reach a larger number of potential end-users. The services that are provided by the service provider can be service logic or content, or both. The service provider can also be compared to a wholesaler.

The TSAS specification is a domain facility enabling end-users to access telecommunication services according to their own wishes. In addition, the specification describes how services can be retailed on behalf of service providers, which in turn offer their services to the retailer.

1.3 User Provider Relationship

TSAS offers mechanisms to establish and release authenticated connections between different domains; therefore, each domain provides interfaces to do so. TSAS uses the terms *user* and *provider* instead of the *client* and *server* terminology, which would be misleading in a number of situations. The user is the role directed to use the interface and provider is the role providing the interface, which is shown in Figure 1-2.

The active role is always the *user role* that initiates the access whereas the passive role is the *provider*, responding to a request.

For a single interaction between two domains request - response user and provider are situated in different *domains*. The domain boundaries are usually based on natural affinities between objects, such as network topology, business stakeholder, or geographical area. In a single scenario more than one user - provider relation may exist (for example, it is possible to have a chain in which a single party acts as a provider in one direction and as user in the other direction).

This may be illustrated by the following example. There is a chain of end-user, retailer, and service provider. The retailer offers services to the end-user, which are realized by the service provider. In the relationship between end-user and retailer the end-user is a user, the retailer a provider. In the relationship between retailer and service provider the retailer is a user and the service provider is a provider.

Note that the definitions in this section imply that the terms *user* and *provider*, *end-user*, and *service provider* have different meanings.

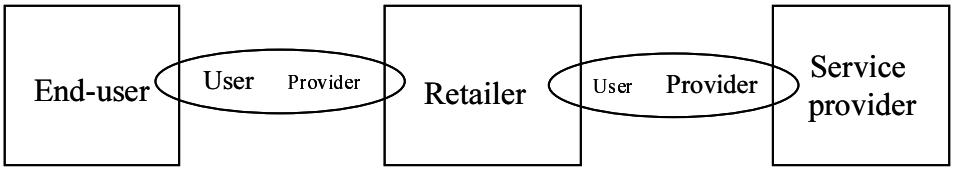

Figure 1-2 use of generic user and provider roles

1.4 Sessions

The usage of services that are implemented taking into account the TSAS framework can be structured in different *sessions*. These are used for grouping specific activities between user and provider. The TSAS specification distinguishes between two different sessions:

- An a*ccess session* is used to establish an authenticated binding between two domains, which in TSAS is between the consumer domain and the retailer domain or between the retailer domain and the service provider domain. It maintains the state about a user's attachment to a provider and about its involvement in services. An access session hence represents the context through which the end-user can access services. The general access session concept also supports all aspects of mobility, that means ubiquitous access by an end-user to the services, irrespective of the terminal being used and the point of attachment to the network.

- A *service session* represents a single activation of a service. It can relate multiple end-users of the service so that they can interact with each other. Moreover, end-users can share resources such as documents or white boards. An end-user may be involved in many services at the same time although it has accessed the retailer only once. The state of a service session is always kept by the service provider (not by the retailer).

Generally, a service cannot be used without having an active access session. Closing the connection between end-user and retailer, or retailer and service provider respectively, will end an access session and also terminate all currently used services.

1.5 Segments

The operations offered by TSAS are grouped in interfaces. The interfaces in turn build *segments*: named sets of interfaces (including so-called callback interfaces) that can be exchanged in one synchronous operation invocation. A segment may consist of two sets of interfaces: one dedicated set for each domain as shown in Figure 1-3.

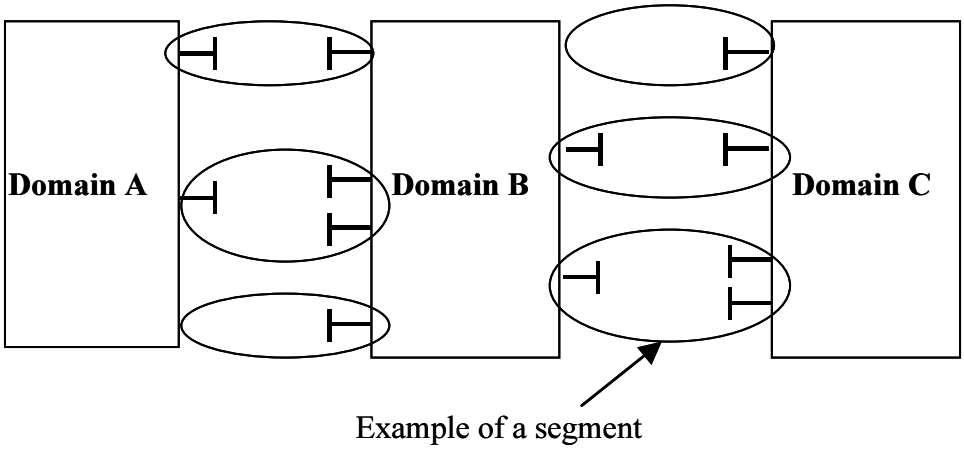

Figure 1-3 Domains, Segments and Interfaces

One TSAS segment is mandatory: the core segment that handles the initial access phase between different domains. This covers the possibility to perform an authentication protocol, and access to services an end-user may wish to use. In addition, it offers the possibility to gain access to other segments supported by the provider.

The other segments can be selected at runtime after an (optional) negotiation phase. Currently these additional segments are defined and described in this chapter and chapter 2: The Invitation Segment, the Context Segment, the Access Control Segment, the Service Discovery Segment, the Session Control Segment, the End-user Customization Segment, the Service Provider Administration Segment, the End-user Administration Segment, and the End-user customization segment They all offer additional service independent functionality.

The usage of optional segments may be tailored for a certain purpose. Segments are self-contained, there exist no dependencies between segments. This eases use of some segments in a certain context, and allows adding additional segments in the future.

The optional segments (also called Service Access Segments and Subscription Segments) are available during an access session only, as described in Section 1.2, “Roles and Domains,” on page 1-2. Its operations allow the end-user or subscriber, retailer, and service provider to interact during an access session in the respective roles of user or provider across domains.

Segments can be requested or supported by the involved domains, depending on the required functionality. Each of these segments can be selected independently of the others. Once selected, however, the segment implementation must use the specifications of this document.

1.6 Security

TSAS uses (mutually) authentication mechanisms between two domains, between the end-user of the consumer domain and the retailer domain, and between the retailer domain and the service provider domain respectively. For authentication either CORBA security can be used or the authentication interface defined in Section 2.2.2, “Authentication Interface,” on page 2-7. Once authenticated, the other optional segments can be used without further authentication for each segment. As a result of the authentication, references of interfaces are available between domains and remain available as long as the relationship resulting from authentication is valid.

Core Segment 2

Contents

This chapter contains the following sections.

2.1 Overview

The core segment is mandatory and defines the interfaces which are used in the initial phase between different domains. This covers the first point of contact to access a provider, the possibility for user and provider to perform an authentication protocol, the access to services they wish to use, and access to other segments supported by the provider.

In TSAS a user contacts a provider to access services offered by the provider. To access these services, the user is required to invoke authentication procedures with the provider before it is able to access services. The use of the terms *user* and *provider* is made according to their definition in the previous chapter.

TSAS defines:

- The first point of contact for a user to access a provider.
- The authentication operations for the user and provider to perform an authentication procedure.
- The user access to services they wish to use.

- The user access to other segments supported by the provider.

The process by which the user accesses the provider has been separated into 3 phases:

1. Initial Contact
2. Authentication
3. Access to the provider's services and segments

Within the core, segment interfaces are defined and within these interfaces operations are defined to enable the user to progress through each of these phases. An overview of these interfaces and operations is given in Figure 2-1.

.

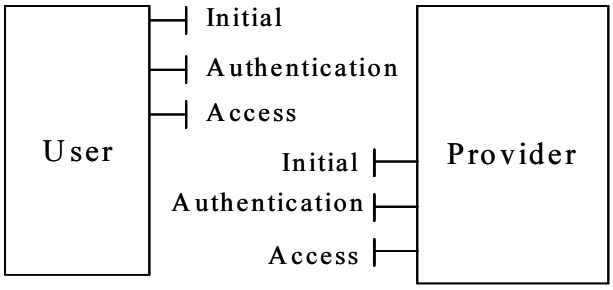

Figure 2-1 Core Interfaces

Initial - This interface allows a user to initiate an authentication procedure and to request access to the provider domain. This initiates an access session; the concept of access session is explained in Section 1.4, "Sessions," on page 1-4. The operations provided are:

- **initiate_authentication()** - allows the user to initiate an authentication procedure.
- **request_access()** - allows the user to request the provider to initiate an access session. If successful the user gains access to an interface for accessing services and other segments offered by the provider.
- **end_access()** - allows the user to terminate an access session.

Authentication - This interface allows a user to proceed through an authentication procedure. It provides the following operations:

- **select_auth_method()** - for selecting the authentication procedure.
- **authenticate()** - to perform the authentication. (It can be invoked several times to complete the authentication procedure).
- **abort_authentication()** - to abort the authentication procedure.

Access - This interface allows an authenticated user to access services and other segments offered by the provider. The interface provides the following operations:

Table 2-1

Operation	Description
list_available_services()	Lists all services that are available at the retailer. The services are scoped using property lists. The operation returns sufficient information for the user to select a service, then start a service
start_session()	To start a service session.
get_service_info ()	Obtain infomation about a service.
end_access()	To end the access session.
end_session()	To end service sessions.
establish_segment()	To set-up a segment.
list_segments()	To list the segments that are available from the provider.
release_segment()	To release a segment.

2.2 Initial Contact and Authentication

Before a user can retrieve information about services offered by a provider, or use these services, they need to contact the provider, and perform an authentication procedure. Figure 2-2 on page 2-4 shows the sequence of operations on the **Initial** and **Authentication** interfaces, for the user to contact the provider, and authenticate. The user then gains access to the **Access** interface to retrieve information on services, use services, and use other interfaces offered by the provider.

- (Before diagram) - User gains a reference to the **Initial** interface of the provider. This may be gained through a URL, an Application Support Broker, a stringified object reference, etc.
- User may invoke **initiate_authentication** on the **Initial** interface. This 'starts' the authentication of the user and provider. The operation allows the user and provider to swap references to the **Authentication** interface. There is the possibility to choose between different authentication types. Here the TSAS authentication type is used also shown in Figure 2-2 the mutual authentication in brackets.
- User invokes **select_auth_method** on the provider's **Authentication** interface. The user identifies to the provider the authentication methods that it can use. Upon return, the provider selects the mechanism that it wishes the user to use.
- User invokes authenticate on the **Authentication** interface, in accordance with the authentication protocol selected. The authenticate operation contains an opaque parameter for the user to fill with data appropriate for the selected authentication protocol. This is the challenge parameter for the provider. The provider is able to

'decode' this parameter, and produce an appropriate response, based upon the challenge data, according to the authentication protocol. This response data is returned to the user in the response parameter. This operation identifies the user unequivocally to the provider.

- The response data is decoded by the user. Depending upon the response data and the selected authentication protocol, the user may need to produce some additional challenge data to the provider. If this is necessary, then the user makes repeated calls using authenticate Authentication. This process continues until the response data indicates that the authentication protocol is complete, the user and provider are satisfied that they have authenticated each other and a credential is passed to the user. If either side is not satisfied with the authentication, they may call the **abortAuthentication** operation to abort the authentication protocol.
- Once user and provider are authenticated, the user invokes the **requestAccess** operation on the **Initial** interface. This operation allows the users to select the type of access that they require. If they select TSAS_ACCESS, then a reference to the TSAS **Access** interface is returned.

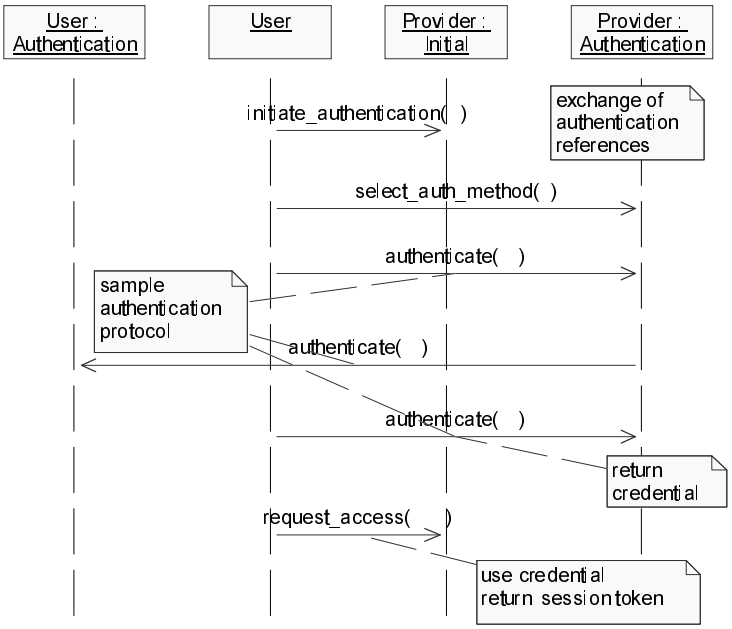

Figure 2-2 Sequence diagram for initial access and authentication

2.2.1 Initial interface

```
interface Initial
{ ...
```

```
};
```

The user gains a reference to the **Initial** interface for the provider that it wishes to access. This may be gained through a URL, an Application Support Broker, a stringified object reference, etc. At this stage, the user has no guarantee that this is a reference to a valid provider.

The user uses this interface to identify himself to the provider and to initiate the authentication process. The **Initial** interface supports the **initiate_authentication** operation to allow the authentication process to take place. It also supports the **request_access** operation to gain access to the provider after the authentication has completed successfully.

2.2.1.1 initiate_authentication()

```
void initiate_authentication (
        in Object user_authentication,
        in AuthType  auth_type,
       out Object provider_authentication)
    raises ( AuthError );
```

The user uses this method to initiate the authentication process.

user_authentication is a reference to an authentication interface at the user domain that can be invoked by the provider to perform the authentication procedure.This interface can be nil, the TSAS **Authentication** interface, or an authentication protocol specific interface. In case of the default TSAS authentication it is a nil reference.

auth_type identifies the type of authentication mechanism requested by the user. It provides users and providers with the opportunity to use an alternative to the **TSAS Authentication** interface (for example, CORBA Security). This authentication process may be specific to the TSAS provider. The **"TSAS_AUTHENTICATION"** provided by the authentication interface is the default authentication method.

If the CORBA Security Service is supported by both the user and the provider, then it may be used to mutually authenticate the user and the provider. The operation of the CORBA security service is out of the scope of TSAS. If it is used to provide authentication of the parties, then the "**CORBA_SECURITY"** value is used for the **auth_type** attribute, and no further authentication is required.

However, if the CORBA Security Service is not supported by both parties, and if further authentication is required, then the TSAS **Authentication** interface can be used. It is obtained by filling the **auth_type** attribute with the value "**TSAS_AUTHENTICATION"**.

The operation returns the **provider_authentication**, a reference to an authentication interface at the provider domain that has to be invoked by the user to perform the authentication procedure. This interface can be the TSAS **Authentication** interface (default), or an authentication protocol specific interface.

2.2.1.2 request_access()

```
request_access (
        in AccessType access_type,
        in Object user_access,
        in Opaque credentials,
        in UserCtxt user_ctxt,
        out Object provider_access,
        out Opaque session_token,
        out AccessSessionId as_id,
        out UserInfo user_info)
    raises ( AccessError, UserCtxtError );
```

The user uses this method to gain access to the provider by means of an access session. This operation must be invoked only after user and provider are authenticated. To guarantee that the user has been authenticated the authentication token **credential** has to be passed to this operation. If this method is called before the user and the provider have successfully completed the authentication process or with an invalid token, then the request fails and an exception AccessError is raised.

access_type identifies the type of access interface requested by the user. Providers can define their own access interfaces to satisfy user requirements for different types of access. If the user requests "**TSAS_ACCESS**", then the TSAS **Access** interface is returned. **TSAS_ACCESS** is the default access method. Depending on the requested access type, the access interface with the corresponding type is returned (see below).

user_access provides the reference for the provider to call the access interface of the user. If the interface reference does not correspond to the type expected, due to the value of **access_type**, an exception AccessError is raised by the provider. This interface reference can be nil.

The **user_ctxt** allows the user to inform the provider about the configuration of his domain. In the particular case of the end-user, it can inform the service provider, via the retailer, of user applications available in the consumer domain, operating systems, etc. **user_ctxt** is a structure containing consumer domain configuration information. If there is a problem with **user_ctxt**, then UserCtxtError should be raised with the appropriate error code.

The returned object **provider_access** provides the reference for the user to call the access interface of the provider. The **session_token** contains an internal identification of the established access session. This token has to be passed as an input parameter to all access related operation. The introduction of the **session_token** allows the provision of one **provider_access** interface per access session or the provision of a common interface per user. Every time an illegal **session_token** is used as an in-parameter a SessionError exception with the value InvalidSessionToken is raised. The access session identification **as_id** can be used to obtain information about all active or resumed service session in an arbitrary access session of the user. This id is given to other access session of the user.

The **user_info** contains information about the user himself. This structure contains the user's UserId, its name, and a list of user properties. Currently no specific property names and values have been defined for **UserPropertyList**, and so its use is provider specific.

2.2.1.3 *end_access ()*

```
end_access (
        in Opaque session_token,
        in EndAccessSessionOption option)
    raises (  SessionError, AccessError  );
```

The user uses this method to terminate an established access session that is identified by the **session_token.** Corresponding to component systems the termination of an access session is provided by the initial interface (factory) and by the **provider_access** interface (instance).

The following behavior during the termination of an access session is possible, depending on the chosen **EndAccessSessionOption**. If the default option is chosen and some active session exits, an AccessError exception with the value ActiveSessions is raised.

Table 2-1 End access session options

DefaultOption	Termination is only possible, if no active service session exists
SuspendActiveSessions	Suspend all active service session and terminate
EndActiveSessions	End all active sessions and terminate
EndAllSessions	End all service sessions and terminate

2.2.2 *Authentication Interface*

```
interface Authentication
{ ...
};
```

Once the user has made initial contact with the provider, authentication of the user and provider may be required. The user may be required to authenticate with the provider before it will be able to use any of the other interfaces supported by the provider. Invocations on other interfaces may fail until authentication has been successfully completed.

TSAS supports several authentication methods. TSAS also defines its own generic authentication mechanism. If the user wants to use the TSAS generic authentication, then it uses the **initiate_authentication** operation on the provider's **Initial** interface

as described above, with **auth_type** parameter set to "**TSAS_AUTHENTICATION"**. The reference returned is the **TSAS Authentication** interface. This interface can be used to support an authentication procedure.

1. The user invokes the **select_auth_method** operation on the provider's **Authentication** interface. This includes the authentication capabilities of the user (that is, the authentication procedures known by the user application). The provider then chooses an authentication procedure based on the authentication capabilities of the user and the provider. If the user is capable of handling more than one authentication procedure, then the provider chooses one option, the **selected_cap**. In some instances, the authentication capability of the user may not fulfill the demands of the provider, in which case, the authentication will fail.

2. The user and provider interact to authenticate each other. Depending on the authentication capability selected, this procedure may consist of a number of interactions (for example, a challenge/response protocol). This authentication procedure is performed using the authenticate operation on the **TSAS Authentication** interface. Depending on the authentication capability selected, the procedure may require invocations on the **Authentication** interface supported by the provider; or on the **Authentication** interface supported by the user; or on both interfaces.

After the authentication procedure has been completed, the user can invoke the **request_access** operation on the **Initial** interface to gain access to the provider's services and other TSAS segments supported by the provider.

2.2.2.1 select_auth_method()

```
void select_auth_method (
        in AuthCapabilityList auth_caps,
       out AuthCapability selected_cap)
    raises ( AuthError );
```

The user invokes the **selectAuthMethod** on the provider's **Authentication** interface to initiate the TSAS generic authentication process. This provides the authentication capabilities of the user to the provider. The provider then chooses an authentication method based on the authentication capabilities of user and provider. The operation returns the selected method (**selected_cap**). In some instances, the authentication capability of the user may not fulfil the demands of the provider, in which case the authentication will fail (the operation raises the exception Authentication Error).

- **auth_caps** is the means by which the authentication mechanisms supported by the user are conveyed to the provider. Examples for authentication capabilities may be (for example, bio ID techniques, chip cards, or username/password combinations).
- **selected_cap** is returned by the provider to indicate the mechanism preferred by the provider for the authentication process among the ones supported by the user that were specified in **authCaps**. If the value of the **selectedCap** returned by the provider is not understood by the user, it should be considered as an unrecoverable error ('panic') and the user should abort its application.

2.2.2.2 *authenticate()*

```
void authenticate (
        in AuthCapability selected_cap,
        in Opaque challenge,
        out Opaque response,
        out Opaque credentials)
    raises ( AuthError );
```

The user and provider use this operation to authenticate each other. It is used according to the authentication procedure, selected by the **selected_cap** parameter (returned by **select_auth_method()**). This procedure may consist of a number of messages (for example, a challenge/ response procedure). The values of the **challenge** and **response** parameters are defined by the authentication procedure. The challenge is used to identify a user uniquely. It may contain a userId or a certificate, which can identify the user by a distinguished name conforming to X.509 v3. In case of a successful authentication the **credentials** parameter contains a token that has to be used as an input parameter in further access operations.

An AuthError exception is raised if the **selected_cap** does not correspond to the **selected_cap** returned by **select_auth_method()**. An AuthError exception is also raised if the challenge data does not correspond to the procedure selected (that is, the challenge data cannot be decrypted according to that method).

The response attribute provides the response of the provider to the challenge data of the user in the current sequence. The response will be based on the challenge data, according to the procedure selected by **select_auth_method ()**.

2.2.2.3 *abort_authentication()*

```
void abort_authentication ( )
    raises ( AuthError );
```

The user uses this method to abort the authentication process. This method is invoked if the user no longer wishes to continue with the authentication process (for example, if the provider responds incorrectly to a challenge). If this method has been invoked, calls to the **request_access** operation on the **Initial** interface will raise the AccessError exception until the user has been properly authenticated. It contains no attributes.

2.3 *Access*

Once a user has been authenticated with a provider an access session is established. The user now can gain access to the services and other segments offered by the provider.

The user invokes the **request_access** operation on the **Initial** interface with the required accessType. If it requests TSAS_ACCESS, then a reference to the **Access** interface is returned. (TSAS Providers can define their own access interfaces to satisfy

user requirements for different types of access). The user also provides the provider with a reference to its ‘callback’ interface to allow the TSAS provider to initiate interactions during the access session.

The **Access** interface allows the user to access services offered by the provider and to gain references to other segments. Segments are defined by TSAS in chapter 3. The sequence for accessing the segments is given in Figure 2-3. Segments are accessed by using the **list_segments()**, **get_segment()**, and **release_segments()** operations.

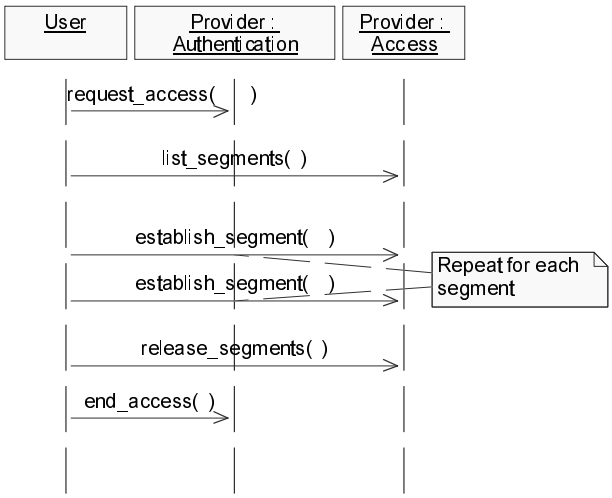

Figure 2-3 Sequence diagram for access segments

list_segments() may be used for getting informed which segments are currently available for a user. With **get_segment ()** a single segment will be returned. This operation needs to be called separately for every segment which shall be used. When segments are not needed anymore, they can be released with **release_segments()**.

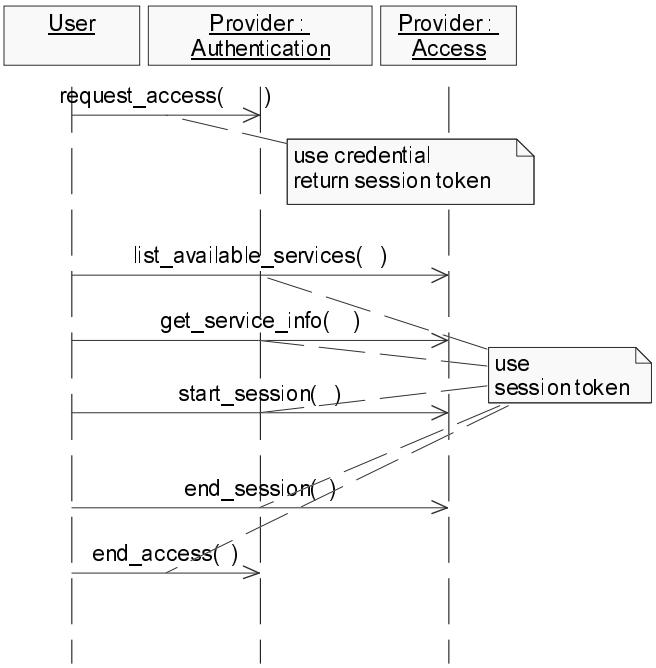

Figure 2-4 Sequence diagram for accessing services

The users use **list_available_services()** to retrieve the **ServiceId** of the service they wish to use. Using the **get_service_info()** operation the user can obtain more information about the service he wants to start. Then the **start_session()** operation is used to initiate the session and return an interface reference to the service. A service session can be ended by using the **end_session** operation. As a result, the interfaces offered by the service are not longer available to the user. The complete process is described in more detail in the following section.

The **end_access** operation is used to end the user's access session with the provider. After it is invoked, the user will no longer be authenticated with the provider. The user will not be able to use the references to any of the provider interfaces gained during the access session. Any calls to these interfaces will fail.

The **Access** interface is also offered by the user to the provider to allow it to initiate interactions during the access session.

2.3.1 Access Interface

```
interface Access
{ ...
};
```

During an authenticated access session the user will be able to select and access services. In order to use a service, the user must be authorized to use the service having establishing a service agreement.

Service agreements can be concluded using either off-line or on-line mechanisms. Off-line agreements will be gained outside of the scope of TSAS interactions and so are not described here. However, users can make use of service agreements that are made off-line. Some providers may only offer off-line mechanisms to conclude service agreements. On-line service agreements may be concluded by using other TSAS provider interfaces, such as the interfaces defined by the subscription segments.

After a service agreement has been established between the user and the provider, the user will be able to make use of this agreement to access a service. The user can use the operations on the **Access** interface to:

- list the services which it can use,
- to start the service session,
- to end the service session, and
- to end the access session.

The **list_available_services()** operation is used to provide a list of services, which the user can use. The user can specify a list of properties that the service must match in order to scope the range of services returned.

The user starts the service session by using the **start_session()** operation. This operation uses the service token to identify the service, with specific service properties, from which to create a new service session. The operation returns a **SessionInfo** structure that contains the **SessionId**, **SessionPropertyList**, and an **InterfaceList** with references to interfaces offered by the service session implementation.

2.3.1.1 list_available_services()

```
void list_available_services (
        in Opaque session_token,
        in ListedServiceProperties desired_properties,
       out ServiceList service_list)
    raises ( SessionError, PropertyError, ListError );
```

The **list_end_user_services()** returns a list of the services that are immediately available to the user in the current access session. It can be noted that the list that is returned can contain services to which the user is already subscribed, as well as services that are (momentarily) offered for free (for which no subscription is required, see section 6 for details on subscription)

The **desired_properties** parameter can be used to scope the list of services. **desired_properties** identifies the properties that the services must match. For example, such a property can indicate that the services returned in the list must all be currently available. **ListedServiceProperties** also defines whether a service must match one, all or none of the properties (see **MatchProperties** in section Section 5.1.1, “Properties and Property Lists,” on page 5-1). Currently no specific

property names and values have been defined for **ListedServiceProperties** ('available' or 'subscribed' would be a good example though), and so its use is service provider specific.

The list of services that matches the **desired_properties** is returned in the ServiceList. This is a sequence of **ServiceInfo** structures which contain the **service_id**, and a sequence of service properties, **ServicePropertyList**. The **service_id** is associated with a specific service when the service is subscribed.

The value of **service_id** is unique among all the available services, but may be different for different users. The **service_id** value persists for the lifetime of the contractual relation between user and provider concerning this service.

Currently no specific property names and values have been defined for **ServicePropertyList**, and so its use is service provider specific.

If the **desired_properties** parameter is wrongly formatted, or provides an invalid property name or value, the PropertyError exception should be raised. Property names that are not recognized can be ignored if **desired_properties** require that only some, or none of the properties are matched. If the service list is unavailable because the retailer's services are not available, then the operation should raise a ListError exception with the ListUnavailable error code.

The operation delivers a list of the services which the user may use. It can be noted that the list that is returned can contain services that are offered for free (for which no subscription is required).

2.3.1.2 get_service_info()

```
void  get_service_info (
        in Opaque session_token,
        in ServiceId  service_id,
        in MatchProperties  desired_properties,
        out ServicePropertyList service_properties)
    raises ( SessionError, ServiceError, PropertyError );
```

The **get_service_info()** returns information on a specific service, identified by the **service_id**. The **desired_properties** list scopes the information that is requested to be returned. The **service_properties** may contain information that is necessary to launch the service GUI.

2.3.1.3 start_session()

```
void start_session (
    in Opaque session_token,
    in ServiceId service_id,
    in ServicePropertyList service_properties,
    out SessionInfo session_info)
raises (
    ServiceError, SessionError, PropertyError );
```

This operation is used by the user to start a service session for the service with the specified **service_id** in the context of the specified access session (**session_token**).

The **service_properties** can be used to pass some service specific attributes from the user to the provider. These properties can be used to initialize the session related service instance.

The returned **session_info** is a structure containing information about the started service session instance. It includes the **SessionId**, **SessionPropertyList**, and a list of interfaces relating to the service session. The **SessionId** has to be unique in the context of all of the user's access sessions. It is used to end, suspend, and resume (refer to Section 3.2.1, "SessionControl Interface") the established service session.

2.3.1.4 end_access()

```
void end_access (
    in Opaque session_token,
    in EndAccessSessionOption option )
raises ( SessionError, AccessError );
```

This operation is used to end the user's access session with the provider. The user requests that its access session is ended. After it is invoked, the user will no longer be authenticated with the provider. The user will not be able to use the references to any of the provider interfaces gained during the access session. Any calls to these interfaces will fail.

EndAccessSessionOption defines the behavior during the termination of the access session. For details refer to table Table 2-1.

2.3.1.5 end_session()

```
void end_session (
    in Opaque session_token,
    in SessionId session_id )
raises ( SessionError );
```

This operation is used to end a service session. After it is invoked, the service session associated with the **session_id** will have ended and will not be accessible to the user, (that is, the user will no longer be able to use any of the references to the session's usage interfaces).

session_id identifies the session to end. If the **session_id** is invalid, a SessionError exception is raised with an InvalidSessionId error code.

2.3.1.6 list_segments()

```
void list_segments (
    in Opaque session_token,
    out SegmentIdList segment_ids )
raises ( SessionError );
```

This operation is used to list the segments offered by the provider. Segments other than this core segment are optional, and so only a subset of the segments defined by TSAS may be offered by a provider. The **segment_ids** returned by this operation only include segment identifiers to segments that are offered by this provider and are available to this user.

2.3.1.7 establish_segment()

```
void establish_segment (
        in Opaque session_token,
        in SegmentId segment_id,
        in InterfaceList user_refs,
        out InterfaceList provider_refs )
    raises ( SessionError, SegmentError, InterfaceError );
```

This operation is used to establish a segment between the user and the provider.

- **segment_id** identifies the segment to be established. The segment defines a number of interface to be offered by the user and the provider. If the **segment_id** is invalid, the provider raises a SegmentError exception with an InvalidSegmentId error code.
- **user_refs** is a list of interfaces supported by the user. It must include references to all interfaces of the types which are required for this segment on the user side. If a required interface is missing from the list, a SegmentError exception is raised with a RequiredSegmentInterfaceNotSupplied error code, and the interface name is returned. If an interface is not part of the segment interfaces, a SegmentError exception is raised, with an InvalidSegmentInterface error code, and the interface name is returned.

A list of interfaces supported by the segment is returned. It must include references to interfaces of the types which must be supported by the provider for this segment.

2.3.1.8 release_segments()

```
void release_segments (
        in Opaque session_token,
        in SegmentIdList segment_ids )
    raises ( SessionError, SegmentError );
```

This operation is used to release segments that have been established between a user and provider. Once a segment is released, the interfaces associated with the segment cannot be used.

segment_ids is a list of segment identifiers of segments to be released. If a segment identifier is invalid, a SegmentError exception is raised with an InvalidSegmentId error code.

2.3.2 Base Interface

```
interface SegmentBase
{
    void release_segment (
        in Opaque session_token)
    raises ( SessionError );
};
```

This is the definition of the base interface from which the segment interfaces can inherit in order for all of them to support the **release_segment** operation. Once a segment is released by calling this operation on one of its supported interfaces, all the interfaces associated with the segment cannot be used.

In contrast to the factory type **release_segments** operation provided by the access segment the **release_segment** operation is provided from every interface instance.

Session Segments 3

Contents

This chapter contains the following sections.

3.1 Overview

This section describes segments that are defined for controlling access and service sessions. In the scope of the ‘Telecommunication Service Access and Subscription’ (TSAS), inter-domain sessions take place on the one hand between the consumer domain and the retailer domain, and on the other hand between the retailer domain and the service provider domain.

The session segments address two types of functionality:

- functionality dedicated to the control of (inter-domain) access sessions, which in turn is specialized into the control of the access between the
  - consumer domain and the retailer domain, and
  - the retailer domain and the service provider domain.
- functionality related to the control of (inter-domain) service sessions for which the consumer domain invokes the retailer domain, and the retailer invokes in its turn one or more service providers.

The functions dedicated to accessing domains consist of retrieving a list of active access sessions.

The functions dedicated to accessing service sessions are:

- listings of service sessions and services, and
- control of service sessions from the access session (e.g., resume, notify changes, etc.).

Figure 3-1 illustrates the various interfaces offered by each domain.

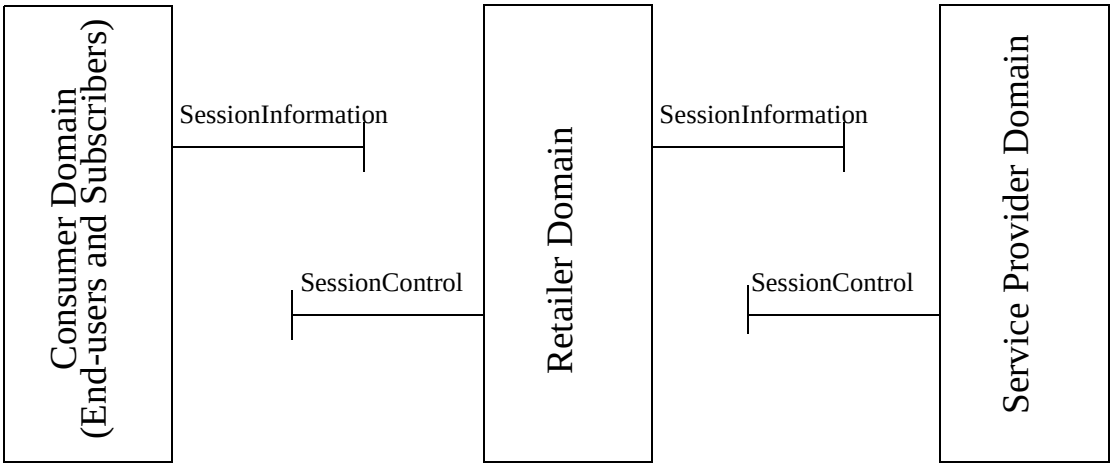

Figure 3-1 Interfaces supported by the TSAS domains

This section globally describes the session segments, their interfaces, and their operations in a generic fashion (that is, for the generic roles of user and provider). This generic specification can be re-used for the specific cases of, on the one hand end-user and retailer, and on the other hand retailer and service provider.

The segments available for use during an access session are:

Session control segment

It provides functionality for service session control. It defines one interface:

1. **SessionControl** - This interface allows a known user to get a list of running service sessions and to resume service sessions or participation in service sessions (when these have been suspended), and to end service sessions.

Session information segment

1. **SessionInformation** - This interface allows a user to receive information over all its access and service sessions from his provider.

These segments, interfaces, and the operations they provide are described below. Since it must be possible to release any segment that is set up from within the segment, all the interfaces inherit from a base interface that defines an operation **release_segment()**. This base interface is defined in the access segment (see Section 2.3.2, “Base Interface,” on page 2-16).

3.2 Session Control Segment

The session control segment defines the **SessionControl** interface.

The **SessionControl** interface allows a known user to get a list of running service sessions and to resume service sessions or participation in service sessions (when these have been suspended).

It provides the following operations:

- **list_access_sessions()** - lists the access sessions of the user.
- **list_service_sessions()** - lists the service sessions of the user. The request can be scoped by the access session and session properties like “active”, “suspended”, or a fixed service type).
- **end_sessions()** - allows the user to end one or more service sessions.
- **suspend_sessions ()** - allows the user to suspend one or more service sessions.
- **resume_session()** - allows the user to resume a service session.

3.2.1 SessionControl Interface

```
interface SessionControl: SegmentBase
{
};
```

The **SessionControl** interface allows a known user to list, to end, and to suspend its running service sessions, and to resume the suspended service sessions. This interface is returned as a result of the **Core::Access::establish_segment()** operation establishing this segment.

3.2.1.1 list_access_sessions()

```
void list_access_sessions (
        in Opaque session_token,
        out AccessSessionIdList as_id_list)
    raises ( SessionError );
```

The **list_access_sessions()** returns a list of access session identifications **AccessSessionIdList** of all access session the user is involved in.

3.2.1.2 list_service_sessions()

```
void list_service_sessions (
        in Opaque session_token,
        in AccessSessionId as_id,
        in SessionSearchProperties desired_properties,
        out SessionDescriptionList session_description_list)
 raises ( SessionError, PropertyError, ListError );
```

The **list_service_sessions()** returns a **SessionList** (list of sessions) of the specified access session **as_id**. This includes active and suspended sessions. A session is associated with an access session if it is being used within that access session, or if it has been suspended in an arbitrary access session.

The **desired_properties** parameter can be used to scope the list of sessions. It identifies the properties that the sessions must match. It also defines whether a session must match one, all or none of the properties (see **MatchProperties** in Section 5.1.1, “Properties and Property Lists,” on page 5-1). The following property names and values have been defined for **SessionSearchProperties**:

If a property in **SessionSearchProperties** has the name “SessionState,” then the matching service session must have the same **SessionState** as given in the property value.

- name: “SessionState”
- value: UserSessionState (“active” or “suspended”)

Other provider specific properties can also be defined in **desired_properties**.

The list of sessions matching the **desired_properties** are returned in sessions. This is a sequence of **SessionDescription** structures which define the **SessionId**, the **SessionState**, and a series of provider specific properties. This information is provider specific, and consequently out of the scope of TSAS.

If the **desired_properties** parameter is wrongly formatted, or provides an invalid property name or value, the PropertyError exception should be raised. Property names that are not recognized can be ignored if **desired_properties** requires that only some, or none of the properties are matched.

If the sessions list is unavailable because the end-user's sessions are not known, then the operation should raise a ListError exception with the ListUnavailable error code.

3.2.1.3 end_sessions()

```
void end_sessions (
        in Opaque session_token,
        in SessionIdList session_id_list)
    raises ( SessionError );
```

The **end_sessions()** ends one or more service sessions, identified by **session_id_list**. The SessionError exception is raised if the **session_token** is invalid or there is an unrecognized **session_id** in the list.

3.2.1.4 suspend_sessions()

```
void suspend_sessions (
        in Opaque session_token,
        in SessionIdList session_id_list)
    raises ( SessionError );
```

The **suspend_sessions()** suspends one or more service sessions, identified by **session_id_list**. The SessionError exception is raised if the **session_token** is invalid or there is an unrecognized **session_id** in the list.

3.2.1.5 resume_session()

```
void resume_session (
        in Opaque session_token,
        in SessionId session_id,
        in ServicePropertyList service_properties,
        out SessionInfo session_info)
    raises ( SessionError, PropertyError );
```

The **resume_session()** resumes a service session. It is used on a service session that is suspended. The suspension and resuming operations are mainly used to obtain service session mobility. The service session can be resumed within an access session different from the one in which the service session was initially running, which possibly involves a different terminal as well. As the operation required to suspend a service session involves service session mobility, the mechanism required to suspend the service session might be service specific, and is therefore not provided by TSAS, but should be defined on one of the service specific interfaces.

session_id - is the identifier of the session to be resumed.

The **service_properties** can be used to pass some service specific attributes from the user to the provider. These properties can be used to re-initialize the session related service instance e.g. to pass some informations about the actual environment of the user.

The returned **SessionInfo** is a structure that contains information that allows the consumer domain to refer to this service session using other operations on this interface. It also contains information for the usage part of the session, including the interface references to interact with the service session (service provider specific).

The exception SessionError is raised if the **session_token** is invalid, the **session_id** is invalid, or if the session refuses to resume because of the user's session state, or if the user does not have permission.

3.2.1.6 Scenario

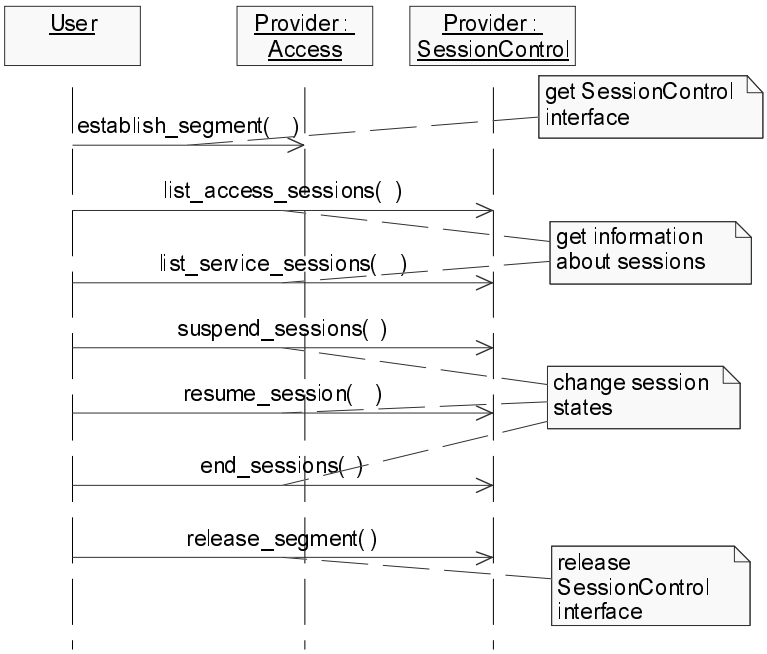

Figure 3-1 Sample Session Control Segment Diagram

The **list_access_sessions(), list_service_sessions()**, **end_sessions()** and **suspend_sessions()** operations are invoked by the user on its provider. The **resume_session()** operation is invoked by the user on its provider for one single service session.

3.3 Session Information Segment

This segment is defined to allow a provider (in the general sense) to inform a user (in the general sense) of changes of state in other access sessions and service sessions with the same user (for example, access sessions with the same user that are created or deleted). The user is only informed about the access sessions he is involved in.

If the user does not want to provide such an interface he is able to obtain information about running sessions by polling the provider's Control Segment.

3.3.1 Session Information Interface

```
interface SessionInformation::SegmentBase
{ ...
};
```

The **SessionInformation** interface allows a user to receive notifications about session changes. This interface is returned as a result of the **Core::Access::establish_segment ()** operation. Using this interface the provider can signal changes of the state of user related sessions. Because these are only notifications they are specified as oneway.

3.3.1.1 new_access_session_info

```
oneway void new_access_session_info (
        in AccessSessionId as_id);
```

The user gets information about a new access session, started with his own user identification. The **as_id** is an identifier for the new access session.

3.3.1.2 end_access_session_info

```
oneway void end_access_session_info (
        in AccessSessionId as_id);
```

The user gets information about a terminated access session The **as_id** is the identifier of the terminated access session.

3.3.1.3 end_session_info

```
oneway void end_session_info (
        in SessionId session_id);
```

The user gets information about a terminated service session The **session_id** is the identifier of the terminated service session.

3.3.1.4 suspend_session_info

```
oneway void suspend_session_info (
        in SessionId session_id);
```

The user gets information about a suspended service session The **session_id** is the identifier of the suspended service session.

3.3.1.5 Scenario

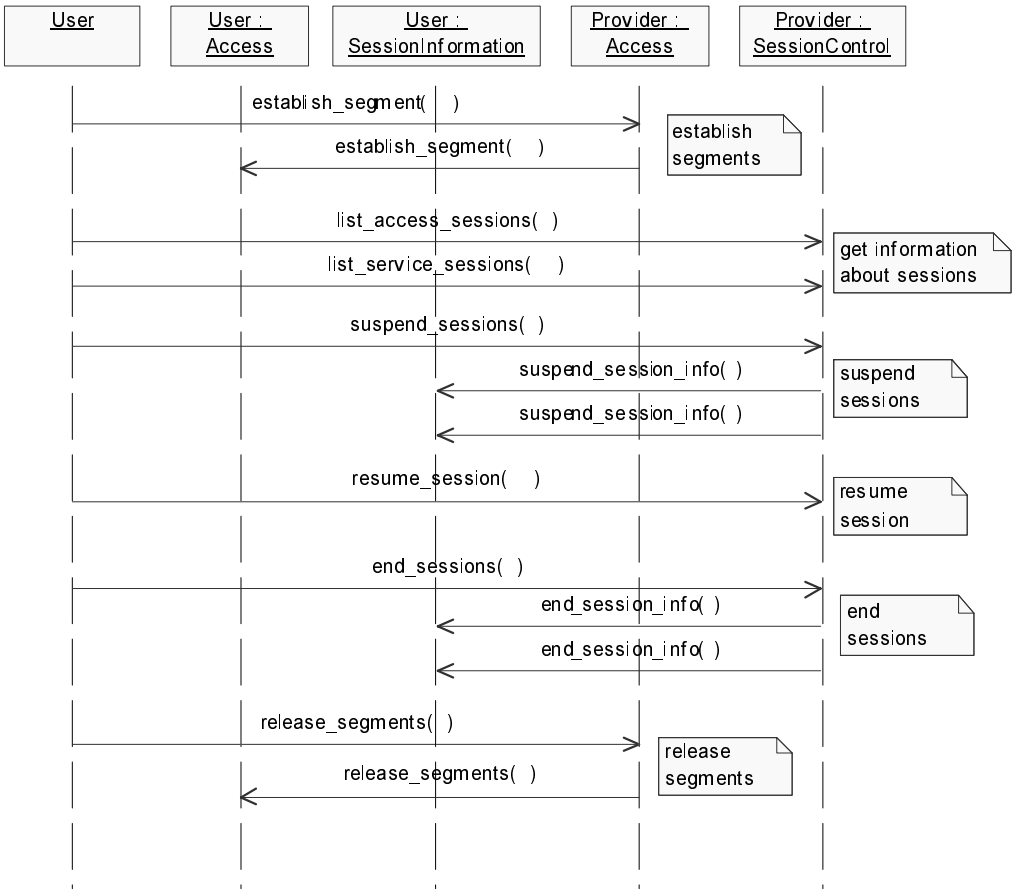

Figure 3-1 Sample Session Information Segment Diagram

Figure 3-1 extends the sample scenario shown in Figure 3-1 . It shows the establishment of the SesionInformation segment and the notifications that are sent from the provider to the user in case of changes of the session states.

Subscription Segments *4*

Contents

This chapter contains the following sections.

4.1 Overview

As described previously in Chapter 3 the retailer mediates services on behalf of service providers to its end-users. The subscription segments offered by the retailer are structured according to the functionality they provide and for which roles (see section Section 2.3.1, “Access Interface,” on page 2-11) they are used, as depicted in Figure 4-1. The subscription segments define interfaces for the consumer domain to be used by end-users and subscribers, and for the service provider domain to be used by service providers.

Subscription manages information about services and contractual relationships between end-user/subscriber and retailer and between service provider and retailer. Before the subscription segments can be used, the end-user/subscriber or the service provider have to access the retailer by establishing an access session as defined in Chapter 2.

All subscription segments described in this chapter provide a framework for the management of subscription information. The retailer can use them to either build an on-line subscription service or use the interfaces to administer its subscription related information. In general the management tasks for subscription encompass management of:

- subscriber related information concerning create, modify and delete subscriber entries.
- service contracts to create, modify or delete service contracts and assign or de-assign service contracts to users.
- user related information concerning the administration of user entries, user groups (subscription assignment groups) and service profiles.
- service templates to deploy, modify or withdraw a service offered by a service provider.

The registration segment provides an operation for the service provider to register itself as subscriber. Following the procedures for accessing the retailer, the subscriber logs in as a user and than registers itself after retrieving the registration segment as a subscriber.

The subscriber administration segment provides interfaces for the management of subscriber related information and can only be user by the subscriber, which has registered itself beforehand by using the registration interface. The subscriber can also manage its users and user groups, called subscription assignment groups.

The service discovery segment is used by subscribers and end-users to discover new services.

The service provider administration segment provides the management of service templates and is used by the service provider.

The end-user customization segment is used by the end-user to manage its personal preferences to be used for customization.

All subscription segments can only be obtained by using the **establish_segment** operation, defined in the core segment (see Chapter 2).

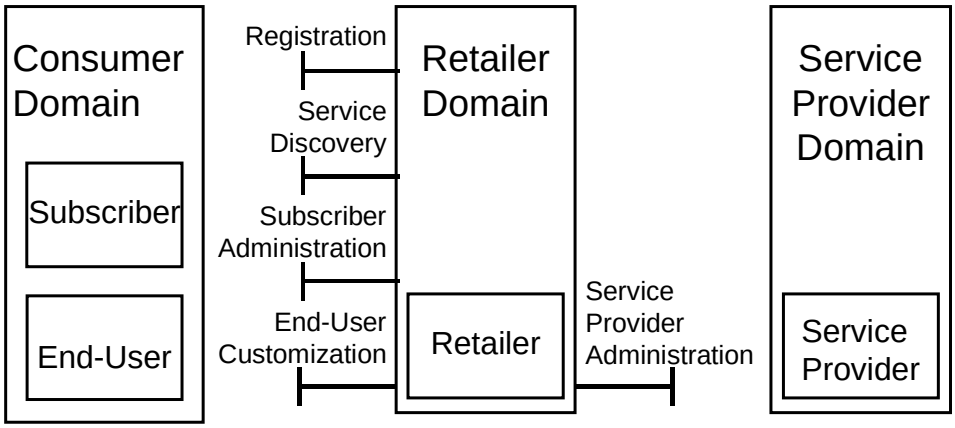

Figure 4-1 subscription Segments provided by the retailer

4.2 Information Model

The information model describes the relationship between information objects that are relevant for the retailer to support subscription segments.

Subscribers play an important role by representing organizations or a single end-user which is going to sign *service contracts* for accessing services provided by a retailer. Signing a service contract gives the permission to use a service under the conditions described in the *service profile*.

If the subscriber represents an organization, it can also manage its *end-users* by creating end-user information objects and building groups of these users, called *subscription assignment groups*. The subscriber will authorize its end-users, groups of users or itself to access services for which it has signed the contract by associating *service profiles* with them. Each service contract refers to a default service profile for the subscribed service. The subscriber can create new service profiles with respect to its contract and associate these service profiles with its end-users and user groups.

The services a retailer provides to the end-users can be services the retailer offers itself or services the retailer offers on behalf of service providers. In the latter case, the service provider also signs a contract with the retailer and registers its services at the retailer domain by using the service provider administration segments.

A *service type* defines the generic classification or category of a service that will be supported by a retailer. The service types are created at the retailer before a particular service corresponding to that type can be deployed by a service provider. The management of service types, that is the creation, modification and deletion of service types, is retailer specific. In general, a retailer (administrator) would decide the types of services it wants to host in its domain. The attributes of the service types are defined by properties, which can be different with respect to each of the different service types. The properties are defined as a list of property name-value pairs. Each attribute has a mode which specifies whether the attribute is mandatory, read-only or normal as defined by Cos Trading Service.

The *service template* contains information about the service attributes, the environment settings, for example configuration information and references to access the service, and application information such as graphical user interface capabilities, language support.

The end-user service profile is related to service specifics which offer the end-user the ability to change individual attributes of the service (for example, to set personal preferences). The end-user service profile is opaque for the retailer and only passed by to the service provider, which interprets it.

The service template conceives the basis for any contractual relationship between a retailer and a subscriber/end-user and between a retailer and a service provider. The service contract itself restricts the range for service settings defined in the service template. The service settings again can be restricted to a subscription assignment group of one subscriber who defines the individual rights for each user or for the subscription assignment groups by setting attributes in the service profile. In addition, a user can specify individual settings within the range of contractual settings predefined by the subscriber in customizable user profiles.

Figure 4-2 illustrates the most relevant information objects using a UML class diagram. The detailed description of these objects and attributes can be found in the text that follows.

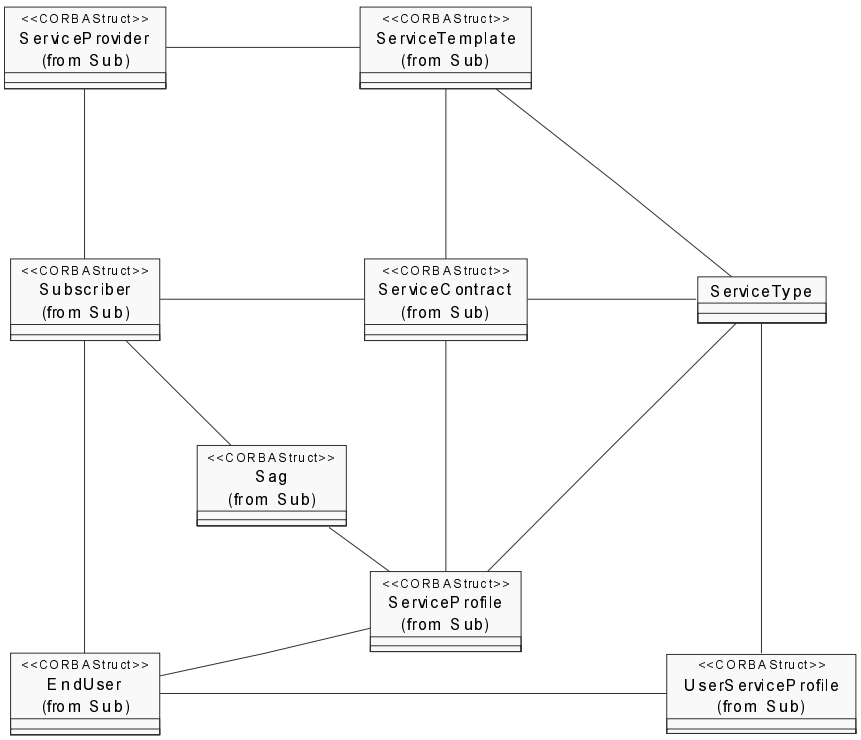

Figure 4-2 Subscription Information Model (example)

4.2.1 Service Provider

```
struct ServiceProvider{
   ProviderId provider_id;
   PropertyList provider_properties;
};
```

The **ServiceProvider** can deploy new services at a retailer by registering service templates of retailer available service types. The service provider object contains a **provider_id** and **provider_properties** which may contain the address, bank account and other details of the service provider.

4.2.2 Subscriber

```
struct Subscriber {
   SubscriberId subscriber_id;
   PropertyList subscriber_properties;
};
```

A **Subscriber** can subscribe to a number of services by signing a service contract. The subscriber object contains a **subscriber_id** and **subscriber_properties**.

Table 4-1 is a non exhaustive example of valuable subscriber properties. The example uses property types as defined in the COS Trading Service. The property mode specifies whether the property is mandatory, read-only, or normal.

Table 4-1 Subscriber Properties

property type	**name**	**type**	**mode**
	first name	string	normal
	last name	string	normal
	orgname	string	normal
	city	string	normal
	street	string	normal
	postal code	string	normal
	email	string	normal
	payment	string	normal

4.2.3 Service Contract

For each subscription a *service contract* exists. The service contract defines the service characteristics for a subscriber and the condition for accessing a service. The service contract properties shall be defined by the retailer.

The *service contract* needs a reference to the *service template* to identify for which service the contract is valid. It also needs the service category defined by the service type for identification. The *service properties* define the specific settings for the service usage by the subscriber. They are a restriction of the *service properties* defined in the *service template*.

```
struct ServiceContract {
            ServiceContractId service_contract_id;
            ServiceTemplateId service_template_id;
            PropertyList service_contract_properties;
            ServiceTypeName service_type;
            PropertyList service_properties;
    };
```

The **service_contract_id** is used to identify the contract. The **service_template_id** provides the reference to the service template. The **service_type** gives the category of the service. The **service_properties** define the setting of the subscriber for the usage of a service.

4.2.4 Service Template

The *service template* defines three kinds of properties:

1. service template properties
2. service properties
3. end-user application properties

```
struct ServiceTemplate{ServiceTemplateId service_template_id;
    ServiceTypeName service_type;
    PropertyList service_template_properties;
    PropertyList service_properties;
    PropertyList user_application_properties;
};
```

The *service_type* defines the category of services a retailer offers.

Table 4-2 illustrates a non exhaustive example of valuable service template properties. The example properties **remote_provider_id**, **remote_initial_agent_naming_context** and **remote_url** are attributes that can be used to provide a reference to access the service provider domain in order to establish an access as defined in Chapter 2.

The example uses property types as defined in the COS Trading Service. The property mode specifies whether the property is mandatory, read-only or normal.

Table 4-2 Service Template Properties

property structure	**name**	**value**	**mode**
	no_start	bool	normal
	depends_on	string	normal
	config_requirements	string	normal
	autostart	bool	normal
	remote_provider_id	string	normal
	remote_inital_agent_name_context	string	normal
	remote_service_id	ulong	normal
	remote_user_id	string	normal
	remote_password	string	normal
	remote_url	string	normal

The **user_application_properties** specify the capabilities of the end-user application. A non exhaustive example of user application properties is given in Table 4-3.

Table 4-3 User Application Properties

property structure	**name**	**value**	**mode**
	default_session_context	string	mandatory
	browser	bool	normal
	orb	string	normal
	java_lib	string	normal
	url	string	normal
	os	string	normal

4.2.5 Subscription Assignment Group

A subscriber may not want to grant all of its end-users the same service characteristics and usage permissions. In this case he can group them into a *Subscription Assignment Group (SAG)* and than assign *service profiles* to each group. The subscriber can also assign more than one service profile for an end-user, for example an internet travel booking service, where each entry page for flight booking, hotel booking or car reservation can be expressed by a separate service profile. *Subscription Assignment Groups (SAG)* are associated with the subscriber.

```
struct Sag{
    SagId sag_id;
    PropertyList properties;
};
```

The **sag_id** is a string defined by the subscriber to identify its SAGs. The subscriber can describe the SAG using the properties. The **sag_id** together with the **subscriber_id** is unique in the retailer system.

4.2.6 Service Profile

The *service profile* specifies the service settings for the usage of that service. It may restrict the *service contract* settings for a specific end user or SAG. It is associated with the *service type*, which relates to the category. The service properties defines the service settings by the subscriber.

```
struct ServiceProfile{
            ServiceProfileId service_profile_id;
            ServiceContractId service_contract_id;
            ServiceTypeName service_type;
            PropertyList service_properties;
    };
```

The **service_profile_id** is used to identify the profile.

The **service_contract_id** provides the reference for which service contract the service profile is valid.

The **service_type** is the corresponding classification given by the retailer.

The **service_properties** specify the specific service settings for a service usage.

4.2.7 End User

An *end-user* will be authorized by a *subscriber* for the access of a service. The *end-user* entry contains an *ID, security_properties* and *user_properties* describing the end-user relevant information for subscription.

```
struct EndUser{
   UserId user_id;
   PropertyList security_properties;
   PropertyList user_properties;
 };
```

The **security_properties** define the kind of authentication a user has, for example password, credential or biometric information. Each end-user can define in the **user_properties** the specific user data such as address, phone number, email.

4.2.8 User Service Profile

The *user service profile* defines the a range of end-user specific possible settings for a customized service usage. The user service profile is related to the service type, which defines the possible properties for end-user specific settings as part of the service properties.

```
struct UserServiceProfile {
   UserServiceProfileId user_service_profile_id;
   ServiceTypeName service_type;
   PropertyList user_service_properties;
 };
```

The **user_service_profile_id** is used to identify the profile. The **user_service_properties** are service dependent and have to be provided by the service provider. The end-user can set its preferences as predefined by the service type which contains the possible range of end-user preferences for a service. The **service_type** is the corresponding classification given by the retailer.

4.2.9 Service Type

The *service type* describes the service category (for example, a communication service). For each service type one service template exists, but there might be multiple service templates for one service type. The service type is described using properties similar to those defined in COS Trading Service. The properties exist for the service specific characteristics (*service profile*) as well as for the preferences (*user service profile*) which can be set by an end-user.

4.3 Subscription Segments

4.3.1 Overview

The subscription interfaces are only available after successful access to the retailer from either the consumer side to manage subscribers and end-users or from the provider side to manage service templates. If the subscription segments are supported by the retailer, the access of the services relevant for subscription will be controlled. That means subscription is checking the authorization of user for service usage.

The segments available for the subscription process are:

Registration Segment

It provides an interface for a subscriber to register itself by the retailer. After successful registration the subscriber is a know role authorized to manage subscriptions at the retailer.

Subscriber Administration Segment

It allows subscribers to manage their subscription. Five interfaces are provided:

1. **SubscriberMgmt** - this interface is used by the subscriber to create, modify or delete subscriber entries and to retrieve subscriber related information.
2. **ServiceContractManagement** - this interface allows subscribers to create modify and delete service contracts and to get information about its contracts and subscribed services.
3. **SagMgmt** - this interface is used for the management of end-user groups. The subscriber can create, modify and delete new groups (SAGs) and retrieve information about already existing groups.
4. **UserMgmt** - this interface is used for the management of users. The subscriber can create, modify and delete user as well as retrieving information about its users.
5. **AuthorizationMgmt** - this interface is used by the subscriber to authorize its users for using a service. It contains operations to manage service profiles. A subscriber can create service profiles for users and assign these profiles to users groups. All operations to retrieve information about service profiles and assignments are also provided.

Service Discovery Segment

It allows a known user to access information about its subscribed services and to discover new services. One interface is provided:

ServDisc - this interface is used by an end-user to get information about subscribed and new services.

Service Provider Administration Segment

It allows service providers to provide new services in the retailer domain. One interface is provided:

ServiceTemplateMgmt - this allows a service provider to register, modify, or unregister a service in the retailer domain. Operations to retrieve information about service templates are also provided.

End-user Customization Segment

This allows end-users to customize the service within the range of predefined settings. One interface is defined:

UserProfileMgmt - this allows an end-user to modify the user profiles and the user service profile settings and to request information about existing user profile

4.4 Scenario Description

To demonstrate the usage of the interfaces the following UML sequence diagrams provide an example set of interfaces related to roles for which the UML actors are used.

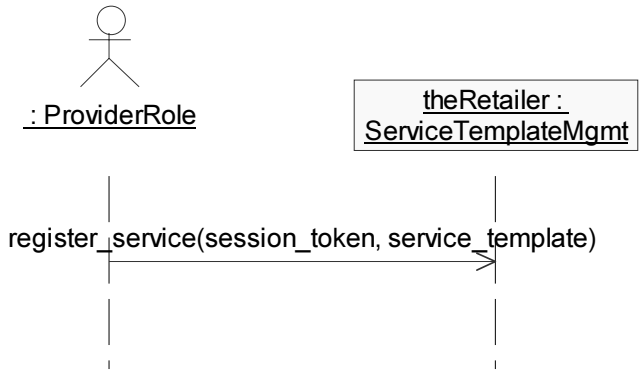

Figure 4-3 Subscription example Scenario for Service Provider Role

Prior to any service usage the retailer needs services that can be used. The service provider registers a new instance of a service template to the retailer by using the **register_service** operation as demonstrated in Figure 4-3. The service provider is a known user of the retailer and has accessed as a user with the role provider. It has to give the service template Id which must be unique in the retailer domain. The service provider sets the service properties, which are provided by the retailer to the end-user.

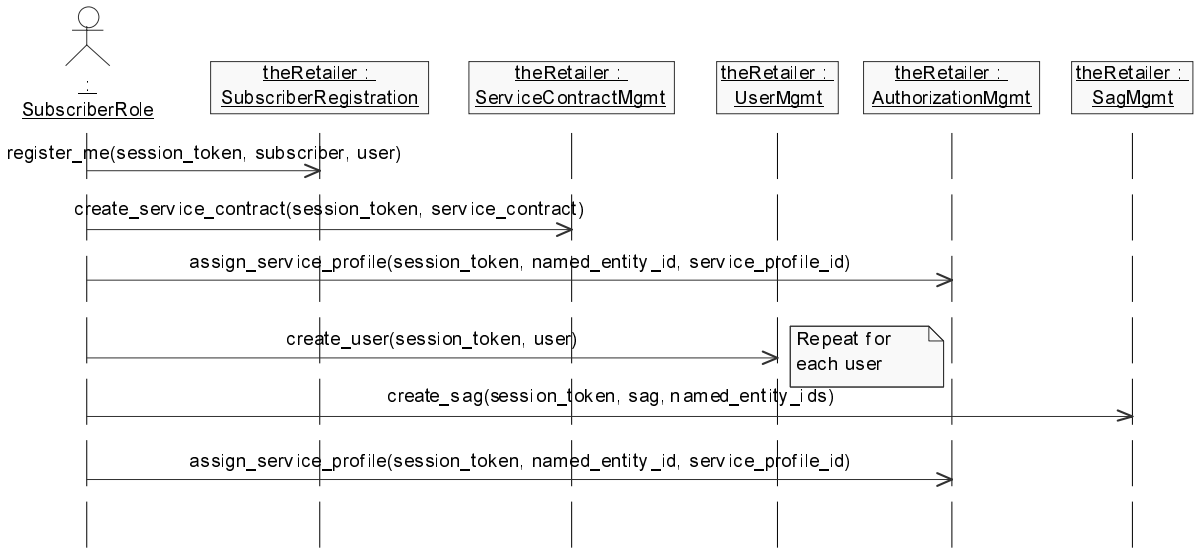

Figure 4-4 Subscription example Scenario for Subscriber Role

1. A subscriber wants to subscribe to a retailer by using the registration segment. A new entry will be created for the subscriber. The subscriber can modify or delete its entries by using the particular operations in the subscriber administration segment.
2. After having an account in the system the subscriber sets up a new contract with the retailer and the required settings for the service. Most of the contract information is defined by a property list, which is defined by retailers.
3. If the subscriber is an end-user it assigns a service profile to define the settings to use a service. After that the service can be used.

1 to 3 are the necessary steps for simple subscription, where a subscriber is also an end-user. The next steps describe the management for end-users.

4. The subscriber creates for each end-user an entry using an user id. The subscriber has to create a new entry for each of its users.
5. The subscriber builds a subscription assignment group and can set the properties of that group. The subscriber adds its users (list of user Ids) to the SAG which he has previously created.
6. The subscriber defines the restrictions for its end-users to access and use a service by setting chosen service properties in the service profile.

.

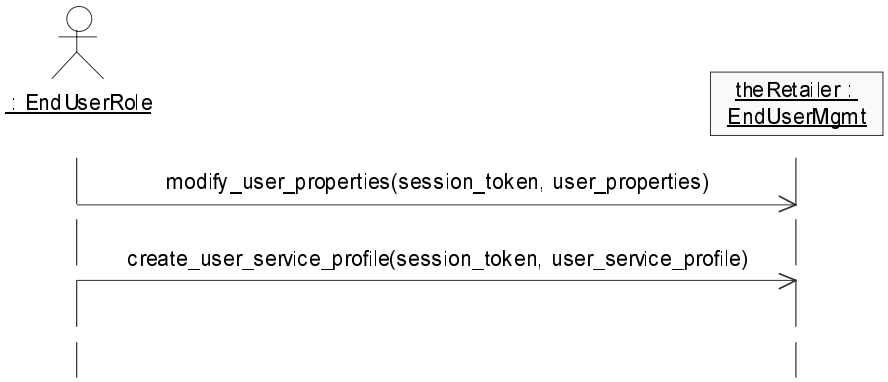

Figure 4-5 Subscription example Scenario for End-User Role

7. The end-user can now edit its user profile and set its preferences.

4.5 Registration Segment

4.5.1 Interface SubscriberRegistration

```
void register_me (
                    in Opaque session_token,
                    in Subscriber subscriber,
```

```
            in EndUser user
    ) raises (
            SubscriptionError, SessionError
    );
```

The operation **register_me** allows an authenticated user to create a new subscriber entry. The subscriber is representing the role and can be either a company or a customer. The **Subscriber** structure contains **subscriber_id** and **subscriber_properties**. The id is given by the subscriber. The **EndUser** represents the user which is the person to be allowed administering the subscription related information. The **EndUser** structure contains **user_id**, **security_properties** and **user_properties**. If the **subscriber_id** or the **user_id** already exists, the operation returns an InvalidSubsciber, InvalidUser or AlreadyExists exception and the subscriber has to try again.

4.6 Subscriber Administration

Subscriber administration segment consists of the management of subscriber entries, the management of service contracts, management of subscription assignment groups, initial management of its users. The Subscriber administration is done by the subscriber.

4.6.1 Subscriber Management

The interface **SubscriberMgmt** is used to define new subscriber entries, to modify and to delete them.

4.6.1.1 interface SubscriberMgmt

```
void
modify_subscriber(
     in Opaque session_token,
     in PropertyList subscriber_properties)
raises (
     SubscriptionError, SessionError
);
```

The operation **modify_subscriber** modifies subscriber entries, for example a new bank account or a new contact person for billing. In the case of invalid **subscriber_properties** the operation returns an InvalidSubscription exception.

```
void
unregister_me(
      in Opaque session_token)
raises (
     SubscriptionError,SessionError
);
```

The **unregister_me** operation removes a subscriber from the system.

```
Subscriber
get_subscriber (
    in Opaque session_token)
raises (
    SubscriptionError, SessionError
);
```

The **get_subscriber** operation returns the information about the subscriber.

4.6.1.2 interface ServiceContractMgmt

The ability to create new service contracts, modify these contracts and delete them is given by the **ServiceContractMgmt** interface.

```
void
create_service_contract(
     in Opaque session_token,
    in ServiceContract  service_contract)
raises (
    SubscriptionError, SessionError
);
```

The operation **create_service_contract** is used by the subscriber to provide the contract relevant information. The service contract_id is used to identify the contract, the service_template_id relates the service contract to the respective service template. The **contract_properties** and the **service_properties** can be set by the subscriber. Exception is InvalidServiceContract.

```
void
modify_service_contract(
     in Opaque session_token,
    in ServiceContract  service_contract)
raises (
    SubscriptionError, SessionError
);
```

The operation **modify_service_contract** is used to modify an existing service contract. The modifications to the contract will be provided by the subscriber. Exception is InvalidServiceContract.

```
void
delete_service_contract(
     in Opaque session_token)
    in ServiceContractId service_contract_id)
raises (
    Subscription Error, SessionError
);
```

The operation **delete_service_contract** removes an existing service contract. The **service_contract_id** identifies the contract. Exception is InvalidServiceContract.

```
ServiceContract
get_service_contract(
     in Opaque session_token,
     in ServiceContractId service_contract_id)
raises (
     SubscriptionError, SessionError
);
```

The Operation **get_service_contract** allows information about a single service contract to be queried and the contract itself to be returned. The **contract_id** is used to identify which contract information shall be provided. Exception is InvalidServiceContract.

```
ServiceContractIdList
list_services(
     in Opaque session_token)
raises (
     SubscriptionError, SessionError
);
```

The **list_services** operation provides a list of all services to which the subscriber has subscribed by a contract. The operation returns a list of subscribed **service_contract_id_list**. Exception is InvalidServiceContract.

4.6.1.3 interface SagMgmt

The **SAGMgmt** interface provides operations to administrate the Subscription Assignment Groups (SAGs) of the subscriber.

```
void
create_sag (
     in Opaque session_token,
     in Sag sag,
     in NamedEntityIdList named_entity_ids)
raises (
     SubscriptionError, SessionError
);
```

For the administration of SAGs the subscriber can use this operation to create a new SAG and to add end-users (which have been created by **create_user**). The **named_entity_ids** are used to reference either **user_ids** or **sag_ids**. The list is given by the subscriber. Exceptions are InvalidSag and InvalidNamedEntityId.

```
void
modify_sag (
     in Opaque session_token,
     in Sag sag)
raises (
     SubscriptionError, SessionError
);
```

The operation **modify_sag** allows a subscriber to modify an existing SAG. Exception is InvalidSag.

```
void
delete_sag (
     in Opaque session_token,
     in SagId sag_id)
raises (
     SubscriptionError, SessionError
);
```

The operation **delete_sag** allows a subscriber to delete an existing SAG. The **sag_id** is used in the retailer domain to identify which SAG should be removed. Exception is InvalidSag.

```
void
add_sag_named_entities(
     in Opaque session_token,
     in SagId sag_id,
     in NamedEntityIdList named_entity_ids)
raises (
     SubscriptionError, SessionError
);
```

A subscriber can add users to specific SAGs by using the operation **add_sag_named_entities**. Before the subscriber can do that, it must have already created the users with the operation create user. The subscriber can use a list with **named_entitiy_ids** to add either user_ids or sag_ids to the subscription assignment group. Exceptions are InvalidSag and InvalidNamedEntityId.

```
void
remove_sag_named_entities(
     in Opaque session_token,
     in SagId sag_id,
     in NamedEntityIdList named_entity_ids)
raises (
     SubscriptionError, SessionError
);
```

The operation **remove_sag_named_entities** removes a single user or a list of users from a SAG of a subscriber. **Sag_id** and **named_entity_ids** are used to identify in the retailer domain which users or sags should be removed. Exceptions are InvalidSag and InvalidNamedEntityId.

```
SagIdList
 list_sags(
     in Opaque session_token)
raises (
     SubscriptionError, SessionError
);
```

The operation **list_sags** allows a subscriber to get a list of already created **sag_ids**.

```
 Sag
 get_sag(
      in Opaque session_token,
      in SagId sag_id)
raises (
      SubscriptionError, SessionError
);
```

The operation **get_sag** allows a subscriber to query information about a single SAG. It returns the sag structure containing **sag_id** and properties. **Sag_id** is used to identify which SAG should be provided to the subscriber. Exception is InvalidSag.

```
NamedEntityIdList
list_sag_named_entities(
      in Opaque session_token,
      in SagId sag_id)
raises (
      SubscriptionError, SessionError
);
```

The operation **list_sag_named_entities** allows a subscriber to get a list of all **named_entity_id_list** containing of user ids or sag ids for a single SAG. The **sag_id** is used in the retailer domain for identification of the SAG. Exception is InvalidSag

4.6.1.4 interface UserMgmt

The user administration interface is intended for situations where an organization wants to allow several end-users to be registered with a retailer. The interface is used by the subscriber, who manages the end-users.

When a registered subscriber wants to provide access to a subscribed service for several end-users in the name of its organization, then it has to register them in the retailer domain.

The main task of the end-user administration is to register, modify and delete user entries.

```
void
create_user(
      in Opaque session_token,
      in EndUser user)
raises (
      SubscriptionError, SessionError
);
```

The operation **create_user** creates a new user. The operation is used by the subscriber. The first entry for an end-user in **end_user** is given by the subscriber, these are the **user _id** and the security properties, whereby the user properties can be defined by the end-user itself by using the end-user customization segment. Exception is InvalidUser.

```
void
modify_user(
    in Opaque session_token,
    in EndUser user)
raises (
    SubscriptionError, SessionError
);
```

The operation **modify_user** modifies information for an existing end-user. The operation is used by the subscriber to modify an user entry. The **user_id** is used to identify for which user the modification should be performed. Exception is InvalidUser.

```
void
delete_user(
    in Opaque session_token,
    in UserId user_id)
raises (
    SubscriptionError, SessionError
);
```

The operations **delete_user** deletes an existing user. The **user_id** is used to identify the user that should be removed. Exception is InvalidUser.

```
EndUser
get_user(
    in Opaque session_token,
    in UserId user_id)
raises (
    SubscriptionError, SessionError
);
```

The operation **get_user** allows a subscriber to query information about a single user. The information about a user contained in the struct **EndUser** is returned. The **user_id** is used to identify which user information shall be provided. Exception is InvalidUser.

```
UserIdList
list_users(
    in Opaque session_token)
raises (
    SubscriptionError, SessionError
);
```

The operation **list_users** returns a list of all **users_ids** of the subscriber.

4.6.1.5 *interface AuthorizationMgmt*

The management of service profiles that are defined for all subscribed services, and the permission for users to use a subscribed service by assigning service profiles to SAGs can be done at the **AuthorizationMgmt** interface. This interface is used by the subscriber.

```
void
create_service_profile(
    in Opaque session_token,
    in ServiceProfile service_profile)
raises (
    SubscriptionError, SessionError
);
```

The operation **create_service_profile** allows a subscriber to create a new service profile. The **profile_id** is given by the subscriber. The service profile contains service parameters which may restrict the service usage. The service profile settings depend on the possibilities the service provider allows and are provided as a list of properties. The subscriber can define different service profiles for one service. Exception is InvalidServiceProfile.

```
void
modify_service_profile(
    in Opaque session_token,
    in ServiceProfile service_profile)
raises (
    SubscriptionError, SessionError
);
```

The operation **modify_service_profiles** allows a subscriber to modify the service profile properties of an already created service profile. Exception is InvalidServiceProfile.

```
void
delete_service_profile(
    in Opaque session_token,
    in ServiceProfileId service_profile_id)
raises (
    SubscriptionError, SessionError
);
```

The operation **delete_service_profiles** allows a subscriber to delete an existing service profile. Exception is InvalidServiceProfile.

```
void
assign_service_profile(
    in Opaque session_token,
    in NamedEntityId named_entity_id,
    in ServiceProfileId service_profile_id)
raises (
```

```
    SubscriptionError, SessionError
);
```

The operation **assign_service_profile** allows a subscriber to assign a service profile to a **named_entity_id**. The previously created service profile will be assigned to a named entity (end user or SAG). Exceptions are InvalidNamedEntityId and InvalidServiceProfile.

```
void
deassign_service_profile(
     in Opaque session_token,
    in NamedEntityId named_entity_id,
    in ServiceProfileId service_profile_id)
raises (
    SubscriptionError, SessionError
);
```

The operation **deassign_service_profile** allows a subscriber to deassign a service profile from a named entity (end user or SAG). Exceptions are InvalidNamedEntityId and InvalidServiceProfile.

```
ServiceProfileIdList
list_service_profiles(
     in Opaque session_token)
raises (
    SubscriptionError, SessionError
);
```

The operation **list_service_profiles** returns a list of all service profiles ids of the subscriber.

```
ServiceProfileIdList
list_assigned_service_profiles(
    in Opaque session_token,
    in NamedEntityId named_entity_id)
raises (
    SubscriptionError, SessionError
);
```

The operation **list_assigned_service_profiles** returns all service profiles assigned to a named entity (end user or SAG). Exception is InvalidNamedEntityId.

```
ServiceProfile
get_service_profile(
    in Opaque session_token,
    in ServiceProfileId service_profile_id)
raises (
    SubscriptionError, SessionError
);
```

The operation **get_service_profile** returns a single service profile. Exception is InvalidServiceProfile.

```
SagIdList
list_assigned_sags(
     in Opaque session_token,
     in ServiceProfileId service_profile_id)
raises (
     SubscriptionError, SessionError
);
```

The operation **list_assigned_sags** returns a list of SAG Ids assigned to single service profile. Exception is InvalidServiceProfile.

```
UserIdList
list_assigned_users(
     in Opaque session_token,
     in ServiceProfileId service_profile_id)
raises (
SubscriptionError, SessionError
);
```

The operation **list_assigned_users** returns a list of **user_ids** that are assigned to single service profile. Exception is InvalidServiceProfile.

4.7 Service ProviderAdministration

The service provider administration segment supports interfaces to service providers to manage the service templates the service provider is going to offer through the retailer domain.

The service template management can be used in order to introduce new services, modify the properties of existing services or delete offered services. The services defined here are actual service offers; for example a video conferencing service of a non-monopolistic telecom operator. The interfaces are used by the service provider.

The service providers can only register those services which are supported by the retailer domain. The retailer itself, in the role of the retailer administrator, decides which kind of services types it supports. Before a service template of a service provider can be registered in the retailer domain, the corresponding service type must be supported. How this is done by the retailer administrator is out of scope of this specification.

At the **ServiceTemplateMgmt** interface the registration, modification and deletion of service templates can be done.

4.7.1 interface ServiceTemplateMgmt

```
     void register_service(
     in Opaque session_token,
     in ServiceTemplate service_template)
raises (
     SubscriptionError, SessionError
```

);

The operation **register_service** allows the service provider to register a new instance of a service template. The service template is given by the service provider. The service provider completes the actual range of provided **service_properties, user_service_properties** and the **service_ template_properties** that are needed for the retailer to access the provider domain to start a service and the possible **end-user_application_properties**, which define the conditions for the **user_application** in the consumer domain. Exception is InvalidServiveTemplate.

How the retailer and service provider exchange the type definitions used in the service template is out of scope of this specification. However, how the retailer defines its own service template that is offered as a service to the end-user is internal to the retailer.

void
modify_service(
in Opaque session_token,
in ServiceTemplate service_template)
raises (
SubscriptionError, SessionError
);

The operation **modify_service** allows a service provider to modify existing service templates (service offers). The **service_template** is a structure containing a service_template_id that is used to identify in the retailer domain which service should be modified. The service capabilities are defined by the service provider in the **service _properties**. Exception is InvalidServiceTemplate.

void
unregister_service(
in Opaque session_token,
in ServiceTemplateId service_template_id)
raises (
SubscriptionError, SessionError
);

The operation **unregister_service** allows a service provider to delete an existing service template. Exception is InvalidServiceTemplate.

ServiceTemplateIdList
list_service_templates(
in Opaque session_token)
raises (
SubscriptionError, SessionError
);

The operation **list_service_templates** returns a list of all service templates of a service provider. The operation returns a list of service template ids.

ServiceTemplate
get_service_template(
in ServiceTemplateId service_template_id)

```
raises (
    SubscriptionError, SessionError
);
```

The operation returns the structure of a single service template. The **service_template_id** is used to identify which service template should be returned. Exception is InvalidServiceTemplate.

4.8 Service Discovery Segment

The service discovery segment defines the **ServiceDiscovery** interface.

The **ServiceDiscovery** interface allows a known user or subscriber to discover new services.

It provides the following operation:

- **discover_services()** - lists all the services available via this retailer (and from the service providers). The user can scope the list by supplying some properties that the service should have, and a maximum number to return.

4.8.1 ServiceDiscovery Interface

```
interface ServiceDiscovery: SegmentBase
{
};
```

This interface is returned as a result of the **Core::Access::establish_segment()** operation establishing this segment.

```
ServiceList
discover_services(
        in Opaque session_token,
        in DiscoverServiceProperties desired_properties,
        in unsigned long how_many)
    raises (
        PropertyError, ListError, SessionError
);
```

The **discover_services()** operation returns a list of the services available via this retailer.This operation is used to discover the services provided via the retailer, for use by the end-user. It can be scoped by the **desired_properties** parameter (see **MatchProperties** in Section 5.1.1, “Properties and Property Lists,” on page 5-1).

The retailer has the possibility to contact one or more service providers in order to fulfill the user's request. This takes place in a way totally transparent to the end-user. The retailer performs one or more invocations on one or more service providers and collects the information received from each service provider. This collected information is merged and provided to the end-user as one piece of information.

The list of retailer services matching the **desired_properties** is returned in services. This is a sequence of **ServiceInfo** structures which contain the **ServiceId** and a sequence of service properties. Currently no specific property names and values have been defined for **DiscoverServiceProperties**, and so its use is service provider specific. Examples of **DiscoverServiceProperties** can be ‘free’ services, ‘comfort’ telephony services, ‘information retrieval’ services, ‘video on demand,’ ‘joint document editing,’ ‘payment,’ ‘calling card reload,’ etc.

The **how_many** parameter defines the number of **ServiceInfo** structures to return in the services parameter. The number of services shall not exceed that number.

If the **desired_properties** parameter is wrongly formatted, or provides an invalid property name or value, the PropertyError exception should be raised. Property names that are not recognized can be ignored if **desired_properties** requires that only some, or none of the properties are matched.

If the services list is unavailable, because the retailer’s services are not available, then the operation should raise an ListError exception with the ListUnavailable error code.

4.8.2 Scenarios

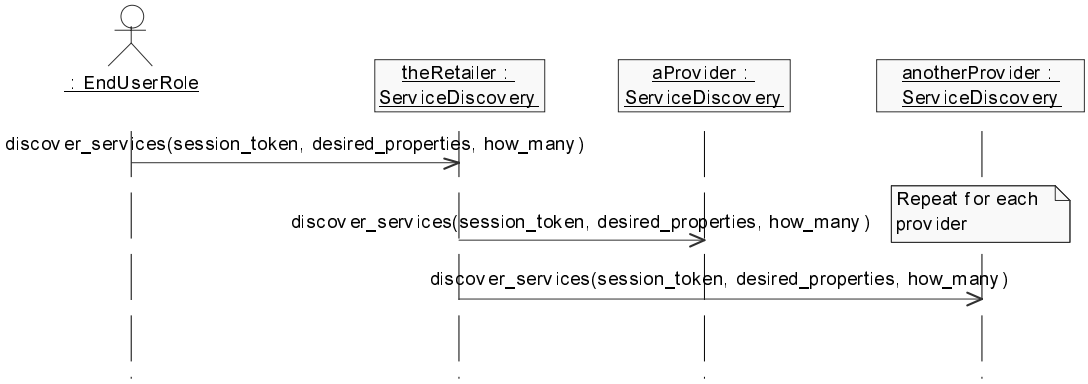

Figure 4-7 Service Discovery Segment Diagram

The **discover_services()** is invoked by the end-user on the retailer. The retailer subsequently invokes **discover_services()** on one or more service provider to fulfil the end-user’s request. The retailer will return the compiled list of discovered services to the end-user.

4.9 End-User Customization

The End-User customization segment allows end-users to customize the service in the range of predefined settings.

The interface **EndUserMgmt** allows an end-user to modify the user profile settings for and the user service profile settings.

4.9.1 interface EndUserMgmt {

```
void modify_security_properties(
      in Opaque session_token,
      in PropertyList  security_properties)
raises (SubscriptionError, SessionError);
```

The operation **modify_security_properties** provides the possibility to change for example the user password. The user password can be one attribute of the user properties. Exception is InvalidProperty.

```
void modify_user_properties(
      in Opaque session_token,
      in PropertyList user_properties )
raises (SubscriptionError, SessionError);
```

The operation **modify_user_profile** allows an end-user to detail its personal entries in the **user_properties**. Exception is InvalidProperty.

```
void create_user_service_profile(
      in Opaque session_token,
      in UserServiceProfile service_profile)
raises (SubscriptionError, SessionError);
```

The operation **create_user_service_profile** provides an end-user with the ability to define the personal preferences for the usage of a service predefined by the service provider. The **service_type_name** is used to identify the related service type. The **user_service_properties** is a property list where the user can define its settings. Exception is InvalidUserServiceProfile.

```
void modify_user_service_profile(
      in Opaque session_token,
      in UserServiceProfile service_profile)
raises (SubscriptionError, SessionError);
```

The operation **modify _user_service_profile** provides an end-user with the ability to change the personal preferences for the usage of a service predefined by the service provider. The **user_service_profile_id** is used to identify the related user service profile which should be modified for the user. The **user_service_properties** is a property list where the user can modify its settings. Exception is InvalidUserServiceProfile.

```
void delete_user_service_profile(
      in Opaque session_token,
      in UserServiceProfileId user_service_profile_id)
raises (SubscriptionError, SessionError);
```

The operation **delete_user_service_profile** removes the user service profile in the retailer domain. The **user_service_profile_id** is used to identify which user service profile shall be deleted. Exception is InvalidUserServiceProfile.

```
EndUser
get_user_description(
    in Opaque session_token)
raises (SubscriptionError, SessionError);
```

The operation **get_user_description** provides the end-user with information about its **user_id** and **user_properties**.

```
UserServiceProfileIdList
list_user_service_profile_ids (
     in Opaque session_token)
raises (SubscriptionError, SessionError);
```

The operation **list_user_service_profiles_ids** returns the list of subscribed end-user service profile identifications.

```
UserServiceProfile
get_user_service_profile(
     in Opaque session_token,
     in UserServiceProfileId user_service_profile_id)
raises (SubscriptionError, SessionError);
);
```

The operation **get_user_service_profile** returns the **end_user_service_profile**. The **service_profile_id** is used to identify the service profile. Exception is InvalidUserServiceProfile.

```
ServiceProfileIdList
get_service_profile_ids(
     in Opaque session_token)
raises (SubscriptionError, SessionError
);
```

The operation **get_service_profile_ids** returns a list of service profile ids for the user.

Common Types 5

Contents

This chapter contains the following sections.

5.1 Generic Information Types

This section describes common types of information which have a high potential for re-use (in several segments, or between other domains than the ones described in the TSAS document).

5.1.1 Properties and Property Lists

Properties are attributes or qualities of something. In TSAS, properties are used to assign a quality to something, or search for items or entities that have that particular quality. The entities that can be qualified by such a property for TSAS can be users, providers, services, sessions, interfaces. Each of these will have different properties, and each property may have a range of different values and structures. While some properties will be defined in this document, some supplementary properties can be defined later and eventually be provider specific.

With this in mind, the type **Property** has been chosen to represent a property. Its IDL definition is taken from the CORBA Property Service. The CORBA type of property names has been changed from string to wstring.

```
typedef wstring PropertyName;
typedef sequence <PropertyName> PropertyNameList;
typedef any PropertyValue;
struct Property {
        PropertyName  name;
        PropertyValue  value;
};
typedef sequence <Property> PropertyList;
```

As can be seen above, the **Property** is a structure consisting of a name and a value. The name is a wstring, and the value is an any. This format allows the recipient of the property to read the wstring and match it against the properties they know about. If it is a property they know, then they will also know the format of the value. If they do not know the property, then they should not read the value. The any value contains a typecode that can be looked up in the interface repository to find the type of the value. The **Property**, and **PropertyList** are used to attribute qualities to entities. Some of these qualities may also be provider-specific, and so they can also use these types to extend the TSAS specifications.

TSAS defines property names and values where it is possible to do so. For some property lists (for example, **InterfaceProperties**) it is up to the user (consumer-/retailer-/service provider domain) to determine properties that can be associated with it.

5.1.2 Match Properties

```
enum WhichProperties {
        NoProperties,
        SomeProperties,
        SomePropertiesNamesOnly,
        AllProperties,
        AllPropertiesNamesOnly
};

struct MatchProperties {
        WhichProperties  which_properties;
        PropertyList  properties;
};
```

MatchProperties is used to scope the return values of some operations. These operations return lists of items. **MatchProperties** is used to identify which items to return, based on the item's properties. For the **operation list_available_services**, the items are a user's subscribed services. The **MatchProperties** parameter defines the properties of the subscribed services that are to be returned in the list.

MatchProperties contains a **PropertyList** and an enumerated type **WhichProperties**. The **PropertyList** contains the properties that need to be matched. The **WhichProperties** identifies whether some, all or none of the properties must be matched, and whether the property name and value, or just the property name must be matched.

For example, in the operation **list_subscribed_services**:

If WhichProperties is...	**Then the subscribed services...**
NoProperties	don't have to match any property, and consequently all subscribed services are returned.
SomeProperties	must match at least one property in the **PropertyList**, (both the property name and value must match), to be included in the returned list.
SomePropertiesNamesOnly	must match at least one property name in the **PropertyList** to be returned. The values of the properties in the **PropertyList** may not be meaningful and should not be used.
AllProperties	must match all the properties in the **PropertyList**, (both the property name and value must match), to be included in the returned list.
AllPropertiesNamesOnly	must match all the property names in the **PropertyList** to be returned. The values of the properties in the **PropertyList** may not be meaningful, and should not be used.

5.2 User Information

```
typedef wstring UserId;
typedef wstring UserName;
typedef PropertyList UserPropertyList;
```

The **user_id** (of type **UserId**) identifies the user to the provider. It is unique to this user within the scope of this provider. The **UserId** does not contain the name of the provider, and so cannot be used to contact the provider. It may be sent to a broker/naming service when attempting to contact a provider along with the provider name.

UserPropertyList is a sequence of **UserProperty**. It contains information about the user that needs to be passed to the provider. The following property names are defined for **UserProperty**. Other property names are allowed, but are provider specific.

```
// Property Names defined for UserPropertyList:
// name:    "PASSWORD"
// value:   string
```

```
// use:      user password, as a wstring.

// name:     "SecurityContext"
// value:    Opaque
// use:      to carry a provider specific security context
// e.g.:     could be used for an encoded user password.
```

5.2.1 UserInfo

This section describes user related information types more dedicated to the access session.

```
struct UserInfo {
        UserId user_id;
        UserName name;
        UserPropertyList user_properties;
};
```

UserInfo describes the end-user. It is a struct of **UserId**, the user's name (that is, readable by a human), and **UserPropertyList**. It is returned by **request_access()** on the **Initial** interface.

5.3 Service Information

```
struct ServiceInfo {
        ServiceId id;
        ServicePropertyList properties;
};
```

ServiceInfo is a structure that describes a subscribed service of the user.

ServiceId is the textual identifier for the service. **ServiceId** is unique among all the user's subscribed services. Other users may be subscribed to the same service, but will have a different **ServiceId**. The **ServiceId** value persists for the lifetime of a subscription.

ServicePropertyList is a property list, which defines the characteristics of this service. They can be used to search for types of service with the same characteristics, (for example, using **discover_services()** on the **ServiceDiscovery** interface of the service discovery segment).

TSAS has defined no properties for **ServicePropertyList**, and so its use is provider specific.

5.4 Access Session Information

```
typedef unsigned long AccessSessionId;
```

The **accessSessionId** of type **AccessSessionId** is used to identify an access session. The **accessSessionId** corresponding to the end-user's current access session is returned at the end of the access session set-up phase. The **accessSessionId** for other access sessions can be found **using list_access_sessions()** in the access control segment, on the **AccessControl** interface. The **AccessSessionId** is scoped by the end-user, (that is, for a single end-user (**UserId**) all **AccessSessionIds** are unique).

5.4.1 User Context Information

The user context information described in this section concerns the user in the generic sense, for example:

- the end-user as a user of the retailer,
- the retailer as a user of the service provider,
- the service provider when it contacts the retailer as a user (for example, for deploying services).

Consequently the user context information can be used in most of the user-provider contexts.

```
typedef wstring UserCtxtName;

typedef PropertyList UserCtxtPropertyList;

struct UserCtxt {
        UserCtxtName  ctxt_name;
        UserCtxtPropertyList  properties;
};
```

UserCtxt informs the provider about the user's environment, including the name of the context.

UserCtxtName is a name given to this user context. It is generated by the user domain. It is used to distinguish between access sessions to different user domains.

Properties is a list of user context related properties that might contain, for example, a list of environment specific attributes.

The UserCtxt is an input parameter of the **request_access()** operation on the **Initial** interface.

5.5 Service Session Information

The session information described in this section concerns the user and provider in the generic sense, for example:

- The end-user as a user of the retailer, in its turn provider to the end-user.
- The retailer as a user of the service provider, in its turn provider to the retailer.

typedef unsigned long SessionId;

All the service sessions running in the service provider domain are identified by a **session_id** (of the type **SessionId**). The retailer must translate these **session_ids**, to provide the consumer with a list of **session_ids** that are unique in the consumer domain. The **SessionId** is a long (32 bits). This **session_id** is the same as the **session_id** provided when a service session is started.

5.5.1 SessionInfo

```
struct SessionInfo {
        SessionId  id;
        SessionPurpose  purpose;
        SessionState state;
        InterfaceList  itfs;
        SessionProperties  properties;
};
```

SessionInfo is a structure that contains information that allows the end-user to refer to a particular service session when using interfaces within an access session. It can also contain information for the usage part of the service session, including the interface references to interact with the service session. The description of these service session interfaces (and their types) is provider specific (outside the scope of TSAS).

Id is the identifier for this service session. It is unique to this service session, among all service sessions that this end-user interacts with through this retailer. If the end-user interacts with multiple retailers concurrently, then they may return **SessionIds** that are identical.

Purpose is a string containing the purpose of the service session. This may have been defined when the service session was created, or subsequently by service specific interactions that are service provider specific.

State is the service session state as perceived by this end-user. It can be: **unknown**, **active**, or **suspended.**

Itfs is a list of interface types and references supported by the service session. It may include service specific interfaces for the user to interact with the service session. Further details are service provider specific.

Properties is a list of properties of the service session. Its use is service provider specific.

OMG IDL *A*

```
#ifndef    _DFTSAS_IDL_
#define    _DFTSAS_IDL_
#pragma    prefix “omg.org”

module DfTsas {

typedef sequence <octet> Opaque;

typedef wstring PropertyName;

typedef sequence <PropertyName> PropertyNameList;

typedef any PropertyValue;

struct Property {
      PropertyName name;
      PropertyValue value;
};

typedef sequence <Property> PropertyList;

enum HowManyProps {
      none,
      some,
      all
};

union SpecifiedProps switch(HowManyProps) {
case some: PropertyNameList prop_names;
};

typedef string InterfaceName;

typedef sequence <InterfaceName> InterfaceNameList;

typedef PropertyList InterfacePropertyList;
```

```
struct InterfaceStruct {
      InterfaceName name;
      Object ref;
InterfacePropertyList properties;
};

typedef sequence <InterfaceStruct> InterfaceList;

typedef wstring Istring;

typedef Istring SegmentId;

typedef sequence <SegmentId> SegmentIdList;

const SegmentId SESSION_INFORMATION_SEGMENT = “Session information”;
const SegmentId ACCESS_CONTROL_SEGMENT = “Access control”;
const SegmentId SERVICE_DISCOVERY_SEGMENT = “Service discovery”;
const SegmentId SESSION_CONTROL_SEGMENT = “Session control”;
const SegmentId SUBSCRIBER_ADMINISTRATION_SEGMENT = “Subscriber administration”;
const SegmentId SERVICE_PROVIDER_ADMINISTRATION_SEGMENT = “Service provider administration”;
const SegmentId END_USER_CUSTOMIZATION_SEGMENT = “End user customization”;
const SegmentId REGISTRATION_SEGMENT = “Customer registration”;

typedef Istring ServiceId;

typedef PropertyList ServicePropertyList;

typedef unsigned long SessionId;

typedef sequence <SessionId> SessionIdList;

typedef Istring SessionState;

typedef Istring SessionPurpose;

typedef PropertyList SessionPropertyList;

struct SessionInfo {
      SessionId id;
      SessionPurpose purpose;
      SessionState state;
      InterfaceList itfs;
      SessionPropertyList properties;
};

typedef PropertyList EndAccessPropertyList;

enum PropertyErrorCode {
      UnknownPropertyError,
      InvalidProperty,
      UnknownPropertyName,
      InvalidPropertyName,
      InvalidPropertyValue,
      NoPropertyError,
```

```
        OtherPropertyError
};

exception PropertyError {
        PropertyErrorCode error;
        PropertyName name;
        PropertyValue value;
};

enum InterfaceErrorCode {
        UnknownInterfaceError,
        InvalidInterfaceName,
        InvalidInterfaceProperty,
        OtherInterfaceError
};

exception InterfaceError {
        InterfaceErrorCode error;
        InterfaceName name;
        PropertyName property_name;
};

enum AuthErrorCode {
        UnknownAuthError,
        InvalidAuthType,
        InvalidAuthCapability,
        NoAcceptableAuthCapability,
        InvalidChallenge,
        OtherAuthError
};

exception AuthError {
        AuthErrorCode error;
        Istring description;
};

enum AccessErrorCode {
        UnknownAccessError,
        InvalidAccessType,
        InvalidAccessInterface,
        AccessDenied,
        ActiveSessions,
        OtherAccessError
};

exception AccessError {
        AccessErrorCode error;
        Istring description;
};

enum ServiceErrorCode {
        UnknownServiceError,
        InvalidServiceId,
        ServiceUnavailable,
        ServiceAccessDenied,
```

```
        OtherServiceError
};

exception ServiceError {
        ServiceErrorCode error;
        Istring description;
};

enum SessionErrorCode {
        UnknownSessionError,
        InvalidUserSessionState,
        SessionNotAllowed,
        SessionNotAccepted,
        InvalidSessionToken,
        OtherSessionError
};

exception SessionError {
        SessionErrorCode error;
        SessionId session_id;
};

enum SegmentErrorCode {
        UnknownSegmentError,
        InvalidSegmentId,
        OtherSegmentError
};

exception SegmentError {
        SegmentErrorCode error;
        SegmentId segment_id;
};

typedef Istring UserId;

typedef Istring UserName;

enum WhichProperties {
        NoProperties,
        SomeProperties,
        SomePropertiesNamesOnly,
        AllProperties,
        AllPropertiesNamesOnly
};

struct MatchProperties {
        WhichProperties which_properties;
        PropertyList properties;
};

typedef MatchProperties ListedServiceProperties;

typedef MatchProperties DiscoverServiceProperties;

typedef MatchProperties SubscribedServiceProperties;
```

```
typedef MatchProperties SessionSearchProperties;

typedef PropertyList UserPropertyList;

struct ServiceInfo {
      ServiceId id;
      ServicePropertyList properties;
};

typedef sequence <ServiceInfo> ServiceList;

typedef unsigned long AccessSessionId;

typedef sequence <AccessSessionId> AccessSessionIdList;

struct UserInfo {
      UserId user_id;
      UserName name;
      UserPropertyList user_properties;
};

typedef Istring UserCtxtName;

typedef PropertyList UserCtxtPropertyList;

struct UserCtxt {
      UserCtxtName ctxt_name;
      UserCtxtPropertyList properties;
};

typedef sequence <SessionInfo> SessionList;

enum ListErrorCode {
      ListUnavailable
};

exception ListError {
      ListErrorCode error;
};

struct PropertyErrorStruct {
      PropertyErrorCode error;
      PropertyName name;
      PropertyValue value;
};

enum UserCtxtErrorCode {
      InvalidUserCtxtProperty,
      OtherUserCtxtError
};

exception UserCtxtError {
      UserCtxtErrorCode error;
      UserCtxtName ctxt_name;
```

```
        PropertyErrorStruct property_error;
};

        struct SessionDescription {
        SessionId session_id;
        SessionState session_state;
        SessionPropertyList session_properties;
};

typedef sequence <SessionDescription> SessionDescriptionList;

enum EndAccessSessionOption {
        DefaultOption,
        SuspendActiveSessions,
        EndActiveSessions,
        EndAllSessions
};

interface SegmentBase {
/*
@roseuid 3CF24D6602AA */
void release_segment (
        in Opaque session_token
        )
        raises (SessionError);

};


module Core {

typedef Istring AuthType;

typedef Istring AccessType;

typedef Istring AuthCapability;

typedef sequence <AuthCapability> AuthCapabilityList;

interface Initial {
        /*
        @roseuid 3CF24D660336 */
        void initiate_authentication (
                in Object user_authentication,
                in AuthType auth_type,
                out Object provider_authentication
                )
                raises (AuthError);

        /*
        @roseuid 3CF24D66033A */
        void request_access (
                in AccessType access_type,
                in Object user_access,
                in Opaque credentials,
```

```
            in UserCtxt user_ctxt,
            out Object provider_access,
            out Opaque session_token,
            out AccessSessionId as_id,
            out UserInfo user_info
            )
            raises (AccessError,UserCtxtError);

        /*
        @roseuid 3CF24D66034A */
        void end_access (
            in Opaque session_token,
            in EndAccessSessionOption option
            )
            raises (SessionError,AccessError);

};

interface Authentication {
        /*
        @roseuid 3CF24D66034D */
        void select_auth_method (
            in AuthCapabilityList auth_caps,
            out AuthCapability selected_cap
            )
            raises (AuthError);

        /*
        @roseuid 3CF24D660356 */
        void authenticate (
            in AuthCapability selected_cap,
            in Opaque challenge,
            out Opaque response,
            out Opaque credentials
            )
            raises (AuthError);

        /*
        @roseuid 3CF24D66035B */
        void abort_authentication ()
            raises (AuthError);

};

interface Access {
        /*
        @roseuid 3CF24D66035E */
        void end_access (
            in Opaque session_token,
            in EndAccessSessionOption option
            )
            raises (SessionError,AccessError);

        /*
        @roseuid 3CF24D660368 */
```

```
void list_available_services (
      in Opaque session_token,
      in ListedServiceProperties desired_properties,
      out ServiceList service_list
      )
      raises (SessionError,PropertyError,ListError);

/*
@roseuid 3CF24D66036C */
void start_sesssion (
      in Opaque session_token,
      in ServiceId service_id,
      in ServicePropertyList service_properties,
      out SessionInfo session_info
      )
      raises (SessionError,ServiceError,PropertyError);

/*
@roseuid 3CF24D660374 */
void end_session (
      in Opaque session_token,
      in SessionId session_id
      )
      raises (SessionError);

/*
@roseuid 3CF24D660377 */
void list_segments (
      in Opaque session_token,
      out SegmentIdList segment_ids
      )
      raises (SessionError);

/*
@roseuid 3CF24D66037C */
void establish_segment (
      in Opaque session_token,
      in SegmentId segment_id,
      in InterfaceList user_refs,
      out InterfaceList provider_refs
      )
      raises (SessionError,SegmentError,InterfaceError);

/*
@roseuid 3CF24D660381 */
void release_segments (
      in Opaque session_token,
      in SegmentIdList segment_ids
      )
      raises (SessionError,SegmentError);

/*
@roseuid 3CF24D660386 */
void get_service_info (
      in Opaque session_token,
```

```
            in ServiceId service_id,
            in MatchProperties desired_properties,
            out ServicePropertyList service_properties
            )
            raises (SessionError,ServiceError,PropertyError);

};

};

module SessCtrl {

interface SessionControl : SegmentBase {
      /*
      @roseuid 3CF24D6603A5 */
      void list_service_sessions (
            in Opaque session_token,
            in DfTsas::AccessSessionId as_id,
            in SessionSearchProperties desired_properties,
            out SessionDescriptionList session_description_list
            )
            raises (SessionError,PropertyError,ListError);

      /*
      @roseuid 3CF24D6603AF */
      void end_sessions (
            in Opaque session_token,
            in SessionIdList session_id_list
            )
            raises (SessionError);

      /*
      @roseuid 3CF24D6603B2 */
      void resume_session (
            in Opaque session_token,
            in SessionId session_id,
            in ServicePropertyList service_properties,
            out SessionInfo session_info
            )
            raises (SessionError,PropertyError);

      /*
      @roseuid 3CF24D6603B8 */
      void suspend_sessions (
            in Opaque session_token,
            in SessionIdList session_id_list
            )
            raises (SessionError);

      /*
      @roseuid 3CF24D6603BB */
      void list_access_sessions (
            in Opaque session_token,
            out AccessSessionIdList as_id_list
            )
```

```
            raises (SessionError);

    };

    interface SessionInformation : SegmentBase {
        /*
        @roseuid 3CF24D6603D7 */
        oneway void new_access_session_info (
            in AccessSessionId as_id
            );

        /*
        @roseuid 3CF24D6603D9 */
        oneway void end_access_session_info (
            in AccessSessionId as_id
            );

        /*
        @roseuid 3CF24D6603DB */
        oneway void end_session_info (
            in SessionId session_id
            );

        /*
        @roseuid 3CF24D6603E1 */
        oneway void suspend_session_info (
            in SessionId session_id
            );

    };

    };

    module Sub {

    enum SubExceptionCode {
        InvalidService,
        InvalidUser,
        InvalidSubscriber,
        InvalidServiceContract,
        InvalidServiceTemplate,
        NotSubscribed,
        InvalidSag,
        InvalidNamedEntityId,
        InvalidServiceProfile,
        InvalidUserServiceProfile,
        InvalidServiceType,
        InvalidSubscription,
        InvalidProperty,
        NotAssigned,
        NotActivated,
        AlreadyExists,
        AlreadyAssigned,
        InternalError,
        OtherSubError
```

```
};

exception SubscriptionError {
      SubExceptionCode reason;
      Istring description;
};

typedef sequence <UserId> UserIdList;

typedef Istring SubscriberId;

typedef sequence <SubscriberId> SubscriberIdList;

typedef Istring ProviderId;

typedef sequence <ProviderId> ProviderIdList;

typedef Istring ServiceTypeName;

typedef Istring ServiceTemplateId;

typedef sequence <ServiceTemplateId> ServiceTemplateIdList;

typedef Istring ServiceProfileId;

typedef sequence <ServiceProfileId> ServiceProfileIdList;

typedef Istring UserServiceProfileId;

typedef sequence <UserServiceProfileId> UserServiceProfileIdList;

typedef Istring ServiceContractId;

typedef sequence <ServiceContractId> ServiceContractIdList;

typedef Istring NamedEntityId;

typedef sequence <NamedEntityId> NamedEntityIdList;

typedef Istring SagId;

typedef sequence <SagId> SagIdList;

struct ServiceProfile {
      ServiceProfileId service_profile_id;
      ServiceContractId service_contract_id;
      ServiceTypeName service_type;
      PropertyList service_properties;
};

struct UserServiceProfile {
      UserServiceProfileId user_service_profile_id;
      ServiceTypeName service_type;
      PropertyList user_service_properties;
};
```

```
struct ServiceTemplate {
      ServiceTemplateId service_template_id;
      ServiceTypeName service_type;
      PropertyList service_template_properties;
      PropertyList service_properties;
      PropertyList user_application_properties;
};

struct ServiceContract {
      ServiceContractId service_contract_id;
      ServiceTemplateId service_template_id;
      PropertyList service_contract_properties;
      ServiceTypeName service_type;
      PropertyList service_properties;
};

struct ServiceProvider {
      ProviderId provider_id;
      PropertyList provider_properties;
};

struct Subscriber {
      SubscriberId subscriber_id;
      PropertyList subscriber_properties;
};

struct EndUser {
      UserId user_id;
      PropertyList security_properties;
      PropertyList user_properties;
};

struct Sag {
      SagId sag_id;
      PropertyList sag_properties;
};

module ServiceProviderAdmin {

      interface ServiceTemplateMgmt : SegmentBase {
            /*
            @roseuid 3CF24D670357 */
            void register_service (
                  in Opaque session_token,
                  in ServiceTemplate service_template
                  )
                  raises (SubscriptionError,SessionError);

            /*
            @roseuid 3CF24D670360 */
            void modify_service (
                  in Opaque session_token,
                  in ServiceTemplate service_template
                  )
```

```
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D670363 */
        void unregister_service (
            in Opaque session_token,
            in ServiceTemplateId service_template_id
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D670366 */
        ServiceTemplateIdList list_service_templates (
            in Opaque session_token
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D67036A */
        ServiceTemplate get_service_template (
            in Opaque session_token,
            in ServiceTemplateId service_template_id
            )
            raises (SubscriptionError,SessionError);

    };

};

module ServDisc {

    interface ServiceDiscovery : SegmentBase {
        /*
        @roseuid 3CF24D67037F */
        ServiceList discover_services (
            in Opaque session_token,
            in DiscoverServiceProperties desired_properties,
            in unsigned long how_many
            )
            raises (PropertyError,ListError,SessionError);

    };

};

module Registration {

    interface SubscriberRegistration : SegmentBase {
        /*
        @roseuid 3CF24D67039D */
        void register_me (
            in Opaque session_token,
            in Subscriber subscriber,
            in EndUser user
            )
```

```
            raises (SubscriptionError,SessionError);

    };

};

module SubscriberAdmin {

    interface SubscriberMgmt : SegmentBase {
        /*
        @roseuid 3CF24D6703B1 */
        void modify_subscriber (
            in Opaque session_token,
            in PropertyList subscriber_properties
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D6703B4 */
        void unregister_me (
            in Opaque session_token
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D6703BB */
        Subscriber get_subscriber (
            in Opaque session_token
            )
            raises (SubscriptionError,SessionError);

    };

    interface ServiceContractMgmt : SegmentBase {
        /*
        @roseuid 3CF24D6703CF */
        void create_service_contract (
            in Opaque session_token,
            in ServiceContract service_contract
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D6703D8 */
        void modify_service_contract (
            in Opaque session_token,
            in ServiceContract service_contract
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D6703DB */
        void delete_service_contract (
            in Opaque session_token,
            in ServiceContractId service_contract_id
```

```
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D6703DE */
        ServiceContract get_service_contract (
            in Opaque session_token,
            in ServiceContractId service_contract_id
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D6703E4 */
        ServiceContractIdList list_services (
            in Opaque session_token
            )
            raises (SubscriptionError,SessionError);

};

interface SagMgmt : SegmentBase {
        /*
        @roseuid 3CF24D68000F */
        void create_sag (
            in Opaque session_token,
            in Sag sag,
            in NamedEntityIdList named_entity_ids
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D68001B */
        void modify_sag (
            in Opaque session_token,
            in Sag sag
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D68001E */
        void delete_sag (
            in Opaque session_token,
            in SagId sag_id
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D680023 */
        void add_sag_named_entities (
            in Opaque session_token,
            in SagId sag_id,
            in NamedEntityIdList named_entity_ids
            )
            raises (SubscriptionError,SessionError);
```

```
        /*
        @roseuid 3CF24D680027 */
        void remove_sag_named_entities (
            in Opaque session_token,
            in SagId sag_id,
            in NamedEntityIdList named_entity_ids
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D68002D */
        SagIdList list_sags (
            in Opaque session_token
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D68002F */
        Sag get_sag (
            in Opaque session_token,
            in SagId sag_id
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D680032 */
        NamedEntityIdList list_sag_named_entities (
            in Opaque session_token,
            in SagId sag_id
            )
            raises (SubscriptionError,SessionError);

};

interface UserMgmt : SegmentBase {
        /*
        @roseuid 3CF24D680055 */
        void create_user (
            in Opaque session_token,
            in EndUser user
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D680058 */
        void modify_user (
            in Opaque session_token,
            in EndUser user
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D68005B */
        void delete_user (
            in Opaque session_token,
```

```
            in UserId end_user_id
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D680060 */
        EndUser get_user (
            in Opaque session_token,
            in UserId user_id
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D680063 */
        UserIdList list_users (
            in Opaque session_token
            )
            raises (SubscriptionError,SessionError);

};

interface AuthorizationMgmt : SegmentBase {
        /*
        @roseuid 3CF24D68007D */
        void create_service_profile (
            in Opaque session_token,
            in ServiceProfile service_profile
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D680080 */
        void modify_service_profile (
            in Opaque session_token,
            in ServiceProfile service_profile
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D680087 */
        void delete_service_profile (
            in Opaque session_token,
            in ServiceProfileId service_profile_id
            )
            raises (SubscriptionError,SessionError);

        /*
        @roseuid 3CF24D68008A */
        void assign_service_profile (
            in Opaque session_token,
            in NamedEntityId named_entity_id,
            in ServiceProfileId service_profile_id
            )
            raises (SubscriptionError,SessionError);
```

```
            /*
            @roseuid 3CF24D680090 */
            void deassign_service_profile (
                in Opaque session_token,
                in NamedEntityId named_entity_id,
                in ServiceProfileId service_profile_id
                )
                raises (SubscriptionError,SessionError);

            /*
            @roseuid 3CF24D680094 */
            ServiceProfileIdList list_service_profiles (
                in Opaque session_token
                )
                raises (SubscriptionError,SessionError);

            /*
            @roseuid 3CF24D680096 */
            ServiceProfileIdList list_assigned_service_profiles (
                in Opaque session_token,
                in NamedEntityId named_entity_id
                )
                raises (SubscriptionError,SessionError);

            /*
            @roseuid 3CF24D68009B */
            ServiceProfile get_service_profile (
                in Opaque session_token,
                in ServiceProfileId service_profile_id
                )
                raises (SubscriptionError,SessionError);

            /*
            @roseuid 3CF24D68009E */
            SagIdList list_assigned_sags (
                in Opaque session_token,
                in ServiceProfileId service_profile_id
                )
                raises (SubscriptionError,SessionError);

            /*
            @roseuid 3CF24D6800A1 */
            UserIdList list_assigned_users (
                in Opaque session_token,
                in ServiceProfileId service_profile_id
                )
                raises (SubscriptionError,SessionError);

        };

    };

    module EndUserCustomization {

        interface EndUserMgmt : SegmentBase {
```

```
/*
@roseuid 3CF24D6800C3 */
void modify_security_properties (
      in Opaque session_token,
      in PropertyList security_properties
      )
      raises (SubscriptionError,SessionError);

/*
@roseuid 3CF24D6800CC */
void modify_user_properties (
      in Opaque session_token,
      in PropertyList user_properties
      )
      raises (SubscriptionError,SessionError);

/*
@roseuid 3CF24D6800CF */
void create_user_service_profile (
      in Opaque session_token,
      in UserServiceProfile user_service_profile
      )
      raises (SubscriptionError,SessionError);

/*
@roseuid 3CF24D6800D2 */
void modify_user_service_profile (
      in Opaque session_token,
      in UserServiceProfile user_service_profile
      )
      raises (SubscriptionError,SessionError);

/*
@roseuid 3CF24D6800D7 */
void delete_user_service_profile (
      in Opaque session_token,
      in UserServiceProfileId user_service_profile_id
      )
      raises (SubscriptionError,SessionError);

/*
@roseuid 3CF24D6800DA */
EndUser getUserDescription (
      in Opaque session_token
      )
      raises (SubscriptionError,SessionError);

/*
@roseuid 3CF24D6800DC */
UserServiceProfileIdList listUserServiceProfileIds (
      in Opaque session_token
      )
      raises (SubscriptionError,SessionError);

/*
```

```
            @roseuid 3CF24D6800E1 */
            UserServiceProfile getUserServiceProfile (
                in Opaque session_token,
                in UserServiceProfileId user_service_profile_id
                )
                raises (SubscriptionError,SessionError);

            /*
            @roseuid 3CF24D6800E4 */
            ServiceProfileIdList get_service_profile_ids (
                in Opaque session_token
                )
                raises (SubscriptionError,SessionError);

        };

};

};

};

module IOP {
const ServiceId ACCESS_SESSION_ID = OMG_assigned;
const ServiceId SERVICE_SESSION_ID = OMG_assigned;
};

/* for #ifndef _DFTSAS_IDL_ */

#endif
```

Compliance Points *B*

This specification does contain elements that are intended to become part of the CORBA standard and, thus, would have to be supported by all CORBA ORBs.

The specification provides three compliance points for implementations of Telecommunication Service Access and Subscription (TSAS), namely, Core Segment, Service Access Segment, and Subscription Segment.

B.1 Core Segment Compliance Point

All conforming implementations must support all interfaces that are defined in Chapter 2 and in document telecom/00-02-03 which contains the IDL specification, following the specified semantics.

B.2 Service Access Segments Compliance Point

An implementation may support any segment defined in Chapter 3, but there is no need to support any of the segments.

When segments are implemented they need to be conformant to the specification given in Chapter 3 and in document telecom/00-02-03 which contains the IDL specification.

B.3 Subscription Segments Compliance Point

An implementation may support any segment defined in Chapter 4, but it is not required to support any of the segments.

When segments are implemented they need to be conformant to the specification given in Chapter 4 and in document telecom/00-02-03 which contains the IDL specification.

B.4 Changes to CORBA

The identification of multiple access sessions introduced in Chapter 2 requires an extension to a very small part of the CORBA architecture. This section details those proposed changes. They are made against CORBA 2.3.1, document formal/99-10-07.

B.4.1 Changes to CORBA Specification

The following service context identifiers are added to the list of service contexts in Section 13.6.7.

```
const ServiceId ACCESS_SESSION_ID = XX; // Reserved for TSAS
const ServiceId SERVICE_SESSION_ID = XX; // Reserved for TSAS
```

The reason for defining these two ServiceIds, is so that a server on which a CORBA invocation is performed, can always retrieve sufficient context information from CORBA so that the client that has performed this invocation is uniquely identified. This has to do with the implementation choices: it is possible that one CORBA server implements more than one access session, or more than one service session. These sessions can potentially be used by different CORBA clients. When a CORBA invocation is made on the CORBA server, it must be able to identify the context (access session identification or service session identification) in which this invocation takes place.